XEROX LISP
RELEASE NOTES

XEROX

3102434
Lyric Release
June 1987

XEROX LISP RELEASE NOTES

3102434

Lyric Release

June 1987

This manual is set in Modern typeface with text written and formatted on Xerox Artificial Intelligence workstations. Xerox laser printers were used to produce text masters.

PREFACE TO RELEASE NOTES

The preliminary Lyric Release Notes provide reference material on the Xerox Lisp environment for the Lyric Beta release. You will find the following information in these notes:

- An overview of significant Xerox extensions to the Common Lisp language
- Discussion of how specific Common Lisp features have affected the Interlisp-D language and the Xerox Lisp environment.
- Notes reflecting the changes made to Interlisp-D, independent of Common Lisp, since the Koto release
- Known restrictions to the use of Xerox Lisp

Release Notes Organization

The Lyric Release Notes present information on the entire Xerox Lisp environment.

Chapter 1, Introduction begins you orientation toward the Xerox Lisp environment. It also lists the manual set for Lyric.

Chapter 2, Notes and Cautions, highlights significant changes in the Xerox Lisp environment.

Chapter 3, Common Lisp/Interlisp Integration, discusses how the integration of Common Lisp into the Xerox Lisp environment affects Interlisp features.

Chapter 4, Changes to Interlisp-D Since Koto, is a collection of notes outlining changes that have taken place in Interlisp-D and its environment since the Koto release. These changes are primarily independent of Common Lisp integration.

Chapter 5, Library Modules, is a synopsis of the changes to Lisp Library Modules..

Chapter 6, User's Guides, is a collection of release notes on the 1108 and 1186 User's Guides; *A User's Guide to Sketch*, and *A User's Guide to TEdit*.

Chapter 7, Known Problems, contains information on existing problems in the environment.

Four Appendices contain complete documentation of newly integrated system features.

Appendix A describes Xerox Lisp's new kind of Exec. Appendix B describes the new structure editor, SEdit. Appendix C presents information on ICONW, which has been moved out of the Koto Library and into the system. Appendix D contains complete

information on Free Menu, another former Koto Library package that has been expanded and added to Xerox Lisp.

Chapters 3 and 4 are organized to parallel the *Interlisp-D Reference Manual* as closely as possible.

To make it easy to use these notes with the *Interlisp-D Reference Manual* the following conventions are used:

Information (changes, etc.) is organized by *Interlisp-D Reference Manual* volume and section. The *Interlisp-D Reference Manual* section level headings were maintained in the Release Notes to aid in cross-referencing.

In many instances the changed page is listed in the following manner:

(II:17.6)

How to Use The Release Notes

We recommend that you use these notes with your *Interlisp-D Reference Manual* and the following documents packaged with the Lyric release:

Xerox Common Lisp Implementation Notes

Common Lisp, the Language by Guy Steele

Lisp Library Modules

Documentation Tools

TABLE OF CONTENTS

[This page intentionally left blank]

1. INTRODUCTION

With the Lyric release of Xerox Lisp, Common Lisp, as specified in *Common Lisp the Language* by Guy Steele, becomes part of the standard Lisp sysout. Many extensions to Common Lisp were also developed and introduced, producing Xerox Common Lisp (XCL).

Integrating Common Lisp and Xerox's extensions into the Xerox Lisp environment, while preserving most of the functionality found in Interlisp-D, required major changes to the system. To the experienced Interlisp user perhaps the most reactive, and potentially confusing, addition to the system is the introduction of Common Lisp packages.

If you are unfamiliar with the way Common Lisp packages work you should read the section on packages in the *Xerox Common Lisp Implementation Notes,* "Packages," and *Common Lisp the Language*, Chapter 11, Packages. However to get you started, you will find a brief explanation of some of the simpler implications of the introduction of packages given below.

Basically, packages provide separate name spaces for symbols. With few exceptions every symbol in the environment is "homed" in a package, that is each symbol is owned by a specific package. The separate name spaces prevent the system from misinterpreting the use of identically named symbols, for example, Interlisp's MAPCAR from Common Lisp's MAPCAR. In general, in the Xerox Lisp system, the print names of symbols include a *package qualifier*; a word or an abbreviation, followed by a colon, that represents the package that owns that particular symbol.

The package qualifier for symbols in the INTERLISP package is IL:, and for "pure" Common Lisp symbols, CL: . All Interlisp symbols, and all symbols in this document (unless otherwise specified) are in the INTERLISP package.

When you first start the system you will find yourself in an executive from which most of the symbols you use in Interlisp are inaccessible without package qualifiers. This is because the system starts up in a Xerox *Common Lisp* executive. In this exec, the *current package* is called XCL-USER and Interlisp symbols are not "visible" from the XCL-USER package without package qualifiers. If the current package contains the symbol you want to use, then you don't need to use a package qualifier.

For example, to login from the default exec (called the XCL Exec) you would have to evaluate **(IL:LOGIN)**. The IL: is the package qualifier that identifies the symbol **LOGIN** as being in the package INTERLISP. If you try to evaluate just **(LOGIN)**, i.e., without a package qualifier, while you are in the XCL Exec you will get the error: "undefined car of form LOGIN."

Until you become accustomed to this change you might find it useful to develop the habit of using the Common Lisp function cl:in-package before working in Interlisp. That is you would type to the exec: (in-package 'il) which would allow you to type all Interlisp symbols to the exec without package qualifiers. However, you will then have to use package qualifiers to use Common Lisp functions, like cl:in-package.

Changes to the Environment

The integration of Common Lisp in Xerox Lisp required that many things in the environment be changed. In the next chapter, Notes and Cautions, you will find a list of the most notable changes, with an emphasis on the incompatible ones.

Using the Release Notes

The organization of the release notes parallels that of the *Interlisp-D Reference Manual* (IRM). Within each chapter of the release notes, IRM chapter and section headings are preserved with respect to order and numbering, even when in conflict with changes to the system (e.g., File Manager (new name) v. File Package (old name)). You may find it helpful to go through the release notes and the IRM together, marking the IRM sections that have release notes. Later when you consult the Reference Manual you will know which sections require you to read the analogous section of the release notes.

2. NOTES AND CAUTIONS

In the course of integrating Common Lisp, Xerox's extensions to Common Lisp, and Interlisp-D, incompatible changes were made to the system. Below you will find a listing of the most critical changes that you should take note of. This list represents only a summary and should not be viewed as an alternative to a thorough reading of the release notes and the *Xerox Common Lisp implementation* Notes themselves.

The list is arranged in rough parallel with the *Interlisp-D Reference Manual* rather than any kind of ranking of the changes.

Incompatible Changes To Interlisp

- In the Lyric release, Koto and Lyric cannot both be supported on one machine.
- You must have Services 10.0 installed on your printers to correctly print TEdit files.
- Interlisp DMACROs are not visible to Common Lisp. If a symbol has both a function definition and a DMACRO property, the new Xerox compiler assumes that the DMACRO was intended as an optimizer for the old Interlisp compiler and ignores it.
- The Common Lisp functions found in Common Lisp: The Language, section 25.4.2, "Other environmental Inquiries" (e.g., LISP-IMPLEMENTATION-TYPE) are in the COMMON LISP (CL:) package.
- The system has a new type of Executive, and the ability to spawn multiple Executive processes. The default executive is Xerox Common Lisp, not Interlisp. The old Executive (the "Programmer's Assistant") is still available but will not be in future releases.

  You should be particularly careful in the new Executives when typing file names, as some file name delimiters now have syntactic significance in the new readtables. In particular, the character colon (:) used in NS file server names is a package delimiter in all new Executives, and the version delimiter semi-colon (;) is a comment character in the Common Lisp Executives. If you type a file name in the form of a symbol to an Exec, you must escape the special characters, or use the multiple escape character around the whole name. For example, in a Common Lisp Exec you might type

  {FS\:Me\:Company}<Fred>Stuff.tedit\;3

  or

  |{FS:Me:Company}<Fred>Stuff.tedit;3|,

  which are equivalent, except that the former is read as all upper case (Common Lisp Exec's read case-insensitively). This

caution should also be noted when copy-selecting file names out of a File Browser.

However, it is recommended that you type file names as strings whenever possible, as virtually all system interfaces accept strings instead of symbols. Two notable exceptions are MAKEFILE and TEDIT, which require symbols when naming files.

Of course, these escaping rules apply *only* to file names typed to an Executive (or in general, a Lisp reader). Individual tools that prompt for a file name in general read the name as a string, so escape characters need not (and should not) be typed. In particular, this is true for the prompt windows of TEdit and File Browser, and the prompt for an Init file when a system with no local Init file is started up.

- The system has a new error system, based on the current Common Lisp proposed error standard, replaces the old Interlisp error system.
- The **!EVAL** debugger command no longer exists and the = and -> break commands are no longer supported..
- The function **ERRORN** no longer exists and **ERRORTYPELIST** is no longer supported. See Chapter 3, Common Lisp?Interlisp Integration, section 14.10 "Error List" for Interlisp errors that are no longer supported.
- A new compiler and compiled code format, .DFASL (FASt Loading) files. The old compiler is still available and produces files in the old format, but with extension .LCOM. The old compiler will not be available in future releases.
- Files produced by the Lyric File Manager cannot be loaded into previous releases of the system. Files compiled in Koto cannot be loaded into Lyric.
- SETQ from the exec does not interact with the File Manager, nor does it print (var reset) (except in the "Programmer's Assistant").
- DWIM/CLISP: CLISP infix is no longer fully supported; users should dwimify old Koto code before running it in Lyric. Additionally, WITH constructs using "←" and BIND constructs in the form of an atom A←B need to be dwimified. ▮See the section. . .
- The functions BREAKDOWN and BRKDWNRESULTS as well as the variables, BRKDWNTYPE and BRKDWNTYPES have been removed from the environment. The Lisp Library Module, SPY supersedes BREAKDOWN.
- The file system supports having multiple streams opened on a single file at one time. This means that the input/output functions accept only streams as arguments, not symbols naming files. This has several implications for Interlisp programmers, one being that the function **CLOSEALL** is no longer implemented. See the Chapter 3, Common Lisp/Interlisp Integration, Streams and Files section, for details.

- Windows cannot be used interchangeably with streams in Common Lisp functions. If you need to use a window in the middle of a Common Lisp function, use (IL:GETSTREAM window) to get the associated display stream.
- Loading CPM-format floppies is very slow in Lyric. Moreover, Lyric is the last release in which the loading of CPM-format floppies will be supported.
- The default Interlisp readtable has been slightly modified to be more in spirit with Common Lisp—the characters colon (:), hash (#) and vertical bar (|) have different meaning. The File Manager gives a choice of reader environments in which to write files, and remembers which one was used for each file.
- READ/PRINT consistency: Old Interlisp code that used **READ** and **PRINT** without being careful about using a particular readtable may need to be fixed.
- The Interlisp function **SKREAD** now defaults its readtable argument to the current readtable, viz., the value of ***READTABLE***, rather than **FILERDTBL**.
- FREEMENU and ICONW, formerly Library modules, are now included in the Lisp.sysout
- A new Lisp editor, SEdit and a new editor interface, **ED**. DEdit is now a library module. See Appendix B.
- Revised fonts: There is a new naming convention for font files, and the printer widths files have correct line leading information. Old Koto fonts can still be used, but you are encouraged to start using the new fonts as soon as practicable.
- Image objects are now stored on files in a way that cannot always be read into Koto. [Lyric on the other hand, can read both the Koto & the new formats.] This means, for example, that you may not be able to share TEdit files or sketches with image objects in them between Koto and Lyric.
- The field names for the CURSOR datatype have been changed.
- Masterscope has been removed from the standard environment. If you wish to use it, load the Masterscope Library module.
- Pattern matching is no longer a part of the standard environment. Pattern matching can be found in the Lisp Library Module, Match.
- PRESS fonts are not part of the standard Xerox Lisp environment since PRESS is now a Library Module.
- In Lyric, the Library module TCP/IP does not work on 1186 workstations that have *both* IOPs with part number 140K03030 *and* "old" ROMs. The problem is not with the IOP board per se, rather it's a problem with the IOP's ROMs. If TCP/IP doesn't work on your 1186 you should check your IOP board revision. If you have the old IOP you may need to

replace the ROMs before you can use TCP/IP, contact your service representative.

TCP/IP does work with newer IOPs—part number 140K05560.

If you attempt to Teleraid a Lyric sysout from a Koto one you should be aware of the following:

1. All symbols will be read as if they were in the INTERLISP package and you can only type a subset of the IL symbols to it.

2. Teleraid will not understand certain Common Lisp datatypes, such as CHARACTER and strings.

With these caveats, you can still get a fair amount of interesting information.

3. COMMON LISP/INTERLISP-D INTEGRATION

This section provides detailed release notes indicating how Common Lisp affects Interlisp-D in Xerox Lisp. Notes are organized to correspond with the original *Interlisp-D Reference Manual* volumes, and sections within these volumes.

VOLUME I—LANGUAGE

Chapter 2 Litatoms

(2.1)

What Interlisp calls a "LITATOM" is the same as what Common Lisp calls a "SYMBOL." Symbols are partitioned into separate name spaces called packages. When you type a string of characters, the resulting symbol is searched for in the "current package." A colon in the symbol separates a package name from a symbol name; for example, the string of characters "CL:AREF" denotes the symbol AREF accessible in the package CL. For a full discussion, see Guy Steele's *Common Lisp, the Language*.

All the functions in this section that create symbols do so in the INTERLISP package (IL), which is also where all the symbols in the *Interlisp-D Reference Manual* are found. Note that this is true even in cases where you might not expect it. For example, U-CASE returns a symbol in the INTERLISP package, even when its argument is in some other package; similarly with L-CASE and SUBATOM. In most cases, this is the right thing for an Interlisp program; e.g., U-CASE in some sense returns a "canonical" symbol that one might pass to a SELECTQ, regardless of which executive it was typed in. However, to perform symbol manipulations that preserve package information, you should use the appropriate Common Lisp functions (See *Common Lisp the Language*, Chapter 11, Packages and Chapter 18, Strings).

Symbols read under an old Interlisp readtable are also searched for in the INTERLISP package. See Section 25.8, Readtables, for more details.

Section 2.1 Using Litatoms as Variables

(I:2.3)

(BOUNDP *VAR*) [Function]

The Interlisp interpreter has been modified to consider any symbol bound to the distinguished symbol **NOBIND** to be unbound. It will signal an UNBOUND-VARIABLE condition on encountering references to such symbols. In prior releases, the interpreter only considered a symbol unbound if it had no dynamic binding and in addition its top-level value was **NOBIND**.

For most user code, this change has no effect, as it is unusual to bind a variable to the particular value **NOBIND** and still deliberately want the variable to be considered bound. However, it is a particular problem when an interpreted Interlisp function is passed to the function **MAPATOMS**. Since **NOBIND** is a symbol, it will eventually be passed as an argument to the interpreted function. The first reference to that argument within the function will signal an error.

A work-around for this problem is to use a Common Lisp function instead. Calls to this function will invoke the Common Lisp interpreter which will treat the argument as a local, not special, variable. Thus, no error will be signaled. Alternatively, one could include the argument to the Interlisp function in a **LOCALVARS** declaration and then compile the function before passing it to **MAPATOMS**. This has the advantage of significantly speeding up the **MAPATOMS** call.

Section 2.3 Property Lists

(I:2.6)

The value returned from the function **REMPROP** has been changed in one case:

(REMPROP *ATM PROP*) [Function]

Removes all occurrences of the property *PROP* (and its value) from the property list of *ATM*. Returns *PROP* if any were found (**T** if *PROP* is **NIL**), otherwise **NIL**.

Section 2.4 Print Names

(I:2.7)

The print functions now qualify the name of a symbol with a package prefix if the symbol is not accessible in the current package. The Interlisp "PRIN1" print name of a symbol does not include the package name.

(I:2.10)

The **GENSYM** function in Interlisp creates symbols interned in the INTERLISP package. The Common Lisp **CL:GENSYM** function creates uninterned symbols.

(I:2.11)

(MAPATOMS *FN*) [Function]

See the note for **BOUNDP** above.

Section 2.5 Characters

A "character" in Interlisp is different from the type "character" in Common Lisp. In Common Lisp, "character" is a distinguished data type satisfying the predicate **CL:CHARACTERP**. In Interlisp, a "character" is a single-character symbol, not distinguishable from the type symbol (litatom). Interlisp also uses a more

efficient object termed "character code", which is indistinguishable from the type integer.

Interlisp functions that take as an argument a "character" or "character code" do not in general accept Common Lisp characters. Similarly, an Interlisp "character" or "character code" is not acceptable to a Common Lisp function that operates on characters. However, since Common Lisp characters are a distinguished datatype, Interlisp string-manipulation functions are willing to accept them any place that a "string or symbol" is acceptable; the character object is treated as a single-character string.

To convert an Interlisp character code *n* to a Common Lisp character, evaluate (**CL:CODE-CHAR** *n*). To convert a Common Lisp character to an Interlisp character code, evaluate (**CL:CHAR-CODE** *n*). For character literals, where in Interlisp one would write (**CHARCODE** *x*), to get the equivalent Common Lisp character one writes #*x*. In this syntax, *x* can be any character or string acceptable to **CHARCODE**; e.g., #\GREEK-A.

Chapter 4 Strings

(I:4.1)

Interlisp strings are a subtype of Common Lisp strings. The functions in this chapter accept Common Lisp strings, and produce strings that can be passed to Common Lisp string manipulation functions.

Chapter 5 Arrays

Interlisp arrays and Common Lisp arrays are disjoint data types. Interlisp arrays are not acceptable arguments to Common Lisp array functions, and vice versa. There are no functions that convert between the two kinds of arrays.

Chapter 6 Hash Arrays

Interlisp hash arrays and Common Lisp hash tables are the same data type, so Interlisp and Common Lisp hash array functions may be freely intermixed. However, some of the arguments are different; e.g., the order of arguments to the map functions in **IL:MAPHASH** and **CL:MAPHASH** differ. The extra functionality of specifying your own hashing function is available only from Interlisp **HASHARRAY**, not **CL:MAKE-HASH-TABLE**, though the

latter does supply the three built-in types specified by *Common Lisp, the Language*.

Chapter 7 Numbers and Arithmetic Functions

(I:7.2)

The addition of Common Lisp data structures within the Xerox Lisp environment means that there are some invariants which used to be true for anything in the environment that are no longer true.

For example, in Interlisp, there were two kinds of numbers: integer and floating. With Common Lisp, there are additional kinds of numbers, namely ratios and complex numbers, both of which satisfy the Interlisp predicate **NUMBERP**. Thus, **NUMBERP** is no longer the simple union of **FIXP** and **FLOATP**. It used to be that a program saying

```
(if (NUMBERP X)
  then (if (FIXP X)
           then ...assume X is an integer...
         else ...can assume X is floating point...))
```

would be correct in Interlisp. However, this is no longer true; this program will not deal correctly with ratios or complex numbers, which are **NUMBERP** but neither **FIXP** nor **FLOATP**.

Section 7.2 Integer Arithmetic

When typing to a *new* Interlisp Executive, the input syntax for integers of radix other than 8 or 10 has been changed to match that of Common Lisp. Use # instead of |, e.g., #b10101 is the new syntax for binary numbers, #x1A90 for hexadecimal, etc. Suffix Q is still recognized as specifying octal radix, but you can also use Common Lisp's #o syntax.

Chapter 10 Function Definition, Manipulation, and Evaluation

Section 10.1 Function Types

All Interlisp **NLAMBDA**s appear to be macros from Common Lisp's point of view. This is discussed at greater length in *The Xerox Common Lisp Impementation Notes*, Chapter 8, Macros.

Section 10.6 Macros

(**EXPANDMACRO** *EXP QUIETFLG* — —) [Function]

EXPANDMACRO only works on Interlisp macros, those appearing on the MACRO, BYTEMACRO or DMACRO properties of symbols. Use **CL:MACROEXPAND-1** to expand Common Lisp macros and

those Interlisp macros that are visible to the Common Lisp compiler and interpreter.

Section 10.6.1 DEFMACRO

(I:10.24)

Common Lisp does not permit a symbol to simultaneously name a function and a macro. In Lyric, this restriction also applies to Interlisp macros defined by **DEFMACRO**. That is, evaluating **DEFMACRO** for a symbol automatically removes any function definition for the symbol. Thus, if your purpose for using a macro is to make a function compile in a special way, you should instead use the new form **XCL:DEFOPTIMIZER**, which affects only compilation. The *Xerox Common Lisp Implementation Notes* describe **XCL:DEFOPTIMIZER**.

Interlisp DMACRO properties have typically been used for implementation-specific optimizations. They are not subject to the above restriction on function definition. However, if a symbol has both a function definition and a DMACRO property, the new Xerox Lisp compiler assumes that the DMACRO was intended as an optimizer for the old Interlisp compiler and ignores it.

Chapter 11 Stack Functions

Section 11.1 The Spaghetti Stack

Stack pointers now print in the form

```
#<Stackp address/framename>.
```

Some restrictions were placed on spaghetti stack manipulations in order to integrate reasonably with Common Lisp's **CL:CATCH** and **CL:THROW**. In Lyric, "it is an error" to return to the same frame twice, or to return to a frame that has been unwound through. This means, for example, that if you save a stack pointer to one of your ancestor frames, then perform a **CL:THROW** or **RETFROM** that returns "around" that frame, i.e., to an ancestor of that frame, then the stack pointer is no longer valid, and any attempt to use it signals an error "Stack Pointer has been released". It is also an error to attempt to return to a frame in a different process, using **RETFROM**, **RETTO**, etc.

The existence of spaghetti stacks raises the issue of under what circumstances the cleanup forms of **CL:UNWIND-PROTECT** are performed. In Xerox Lisp, **CL:THROW** always runs the cleanup forms of any **CL:UNWIND-PROTECT** it passes. Thanks to the integration of **CL:UNWIND-PROTECT** with **RESETLST** and the other Interlisp context-saving functions, **CL:THROW** also runs the cleanup forms of any **RESETLST** it passes. The Interlisp control transfer constructs **RETFROM**, **RETTO**, **RETEVAL** and **RETAPPLY** also run the cleanup forms in the analogous case, viz., when returning to a direct ancestor of the current frame. This is a

significant improvement over prior releases, where **RETFROM** never ran any cleanup forms at all.

In the case of **RETFROM**, etc, returning to a non-ancestor, the cleanup forms are run for any frames that are being abandoned as a result of transferring control to the other stack control chain. However, this should not be relied on, as the frames would not be abandoned at that time if someone else happened to retain a pointer to the caller's control chain, but subsequently never returned to the frame held by the pointer. Cleanup forms are *not* run for frames abandoned when a stack pointer is released, either explicitly or by being garbage-collected. Cleanup forms are also not run for frames abandoned because of a control transfer via **ENVEVAL** or **ENVAPPLY**. Callers of **ENVEVAL** or **ENVAPPLY** should consider whether their intent would be served as well by **RETEVAL** or **RETAPPLY**, which *do* run cleanup forms in most cases.

Chapter 12 Miscellaneous

Section 12.4 System Version Information

All the functions listed on page 12.12 in the *Interlisp-D Reference Manual* have had their symbols moved to the LISP (CL) package. They are *not* shared with the INTERLISP package and any references to them in your code will need to be qualified i.e., CL:*name*.

Section 12.8 Pattern Matching

Pattern matching is no longer a standard part of the environment. The functionality for Pattern matching can be found in the Lisp Library Module called MATCH.

Chapter 13 Interlisp Executive

[This chapter of the Interlisp-D Reference Manual has been renamed Chapter 13, Executives.]

Xerox Lisp has a new kind of Executive (or Exec), designed for use in an environment with both Interlisp and Common Lisp. This executive is available in three standard modes, distinguished by their default settings for package and readtable:

XCL — New Exec. Uses XCL readtable, XCL-USER package

CL — New Exec. Uses LISP readtable, USER package

IL — New Exec. Uses INTERLISP readtable, INTERLISP package

In addition, the old Interlisp executive, the "Programmer's Assistant", is still available in this release for the convenience of Koto users:

OLD-INTERLISP — Old "Programmer's Assistant" Exec. Uses OLD-INTERLISP-T readtable, INTERLISP package. It is likely that this executive will not be supported in future releases

When Xerox Lisp starts, it is running a single executive, the XCL Exec. You can spawn additional executives by selecting EXEC from the background menu. The type of an executive is indicated in the title of its window; e.g., the initial executive has title "Exec (XCL)". Each executive runs in its own process; when you are finished with an executive, you can simply close its window, and the process is killed.

The new executive is modeled, somewhat, on the old "Programmer's Assistant" executive and, to a first approximation, you can type to it just as you did in past releases. You should note, however, that the default executive (XCL) expects Common Lisp input syntax, and reads symbols relative to the XCL-USER package. This means that to type Interlisp symbols, you must prefix the symbol with the characters "IL:" (in upper or lower case). And even in the new IL executive, the readtable being used is the new INTERLISP readtable, in which the characters colon (:), vertical bar (|) and hash (#) all have different meanings than in Koto.

The OLD-INTERLISP exec, with one exception, uses exactly the same input syntax as in Koto; this means in particular that colon cannot be used to type package-qualfied symbols, since colon is an ordinary character there. The one exception is that there *is* a package delimiter character in the OLD-INTERLISP readtable, should you have a need to use it—Control-↑, which usually echoes as "↑↑", though it may appear as a black rectangle in some fonts.

The new executive does differ from the old one in several respects, especially in terms of its programmatic interface. Complete details of the new executive can be found in [Appendix A. The Exec]. Some of the important differences are:

- Executives are numbered

Executives, other than the first one, are labeled with a distinct number. This number appears in the exec window's title, and also in its prompt, next to the event number. The OLD-INTERLISP executive does not include this exec number.

- Event number allocation

The numbers for events are allocated at the time the prompt for the event is printed, but all execs still share a common event number space and history list. This means that ?? shows all events that have occurred in *any* executive, though not necessarily in the order in which the events actually occurred (since it is the order in which the event numbers were allocated). Events for which the type-in has not been completed are labeled "<in progress>" in the ?? listing. In the old executive, event numbers are not allocated until type-in is complete, which means that the number printed next to the prompt is not necessarily the number associated with the event, in the case that there has been activity in other executives.

In the new executive, relative event specifications are local to the exec; e.g., **-1** refers to the most recent event in that specific exec. In the old executive, **-1** referred to the immediately preceding event in *any* executive.

- New facility for commands

The old Executive has commands based on **LISPXMACROS**. The new Executive has its own command facility, **XCL:DEFCOMMAND**, which allows commands to be named without regard to package, and to be written with familiar Common Lisp style of argument list.

- Commands are typed *without* parentheses

In the old executive, a command could be typed with or without enclosing parentheses. In the new executive, a parenthesized form is always interpreted as an EVAL-style input, never a command.

- **SETQ** does not interact with the File Manager

In the Koto release, when you typed in the Exec

(SETQ FOO *some-new-value-for-FOO*)

the executive responded **(FOO reset)**, and the file package was told that **FOO**'s value changed. Any files on which **FOO** appeared as a variable would then be marked as needing to be cleaned up. If **FOO** appeared on no file, you'd be prompted to put it on one when you ran **(FILES?)**.

This is still the case in the old executive. However, it is no longer the case in the new executive. If you are setting a variable that is significant to a program and you want to save it on a file, you should use the Common Lisp macro **CL:DEFPARAMETER** instead of **SETQ**. This will give the symbol a definition of type **VARIABLES** (rather than **VARS**), and it will be noticed by the File manager. If you want to change the value of the variable, you

must either use **CL:DEFPARAMETER** again, or edit the variable using **ED** (not **DV**).

- Programmatic interface completely different

As a first approximation, all the functions and variables in IRM Sections 13.3 (except the **LISPXPRINT** family) and 13.6 apply only to the Old Interlisp Executive, unless specified otherwise in Appendix A. In particular, the variables **PROMPT#FLG**, **PROMTPCHARFORMS**, **SYSPRETTYFLG**, **HISTORYSAVEFORMS**, **RESETFORMS**, **ARCHIVEFN**, **ARCHIVEFLG**, **LISPXUSERFN**, **LISPXMACROS**, **LISPXHISTORYMACROS** and **READBUF** are not used by the new Exec. The function **USEREXEC** invokes an old-style Executive, but uses the package and readtable of its caller. The function **LISPXUNREAD** has no effect on the new Exec. Callers of **LISPXEVAL** are encouraged to use **EXEC-EVAL** instead.

Some subsystems still use the old-style Executive—in particular, the tty structure editor.

Chapter 14 Errors and Breaks

Xerox Lisp extends the Interlisp break package to support multiple values and the Common Lisp lambda syntax. Interlisp errors have been converted to Common Lisp conditions.

Note that Sections 14.2 through 14.6 in the *Interlisp-D Reference Manual* have been replaced by new Debugger information (see *Common Lisp Implementation Notes)*.

Section 14.3 Break Commands

(II:14.6)

The **!EVAL** debugger command no longer exists.

(II:14.10-11)

The Break Commands = and -> are no longer supported.

Section 14.6 Creating Breaks with BREAK1

While the function **BREAK1** still exists, broken and traced functions are no longer redefined in terms of it. More primitive constructs are used.

Section 14.7 Signalling Errors

Interlisp errors now use the new XCL error system. Most of the functions still exist for compatibility with existing Interlisp code, but the underlying machinery is different. There are some incompatible differences, however, especially with respect to error numbers.

The old Interlisp error system had a set of registered error numbers for well known error conditions, and all other errors were identified by a string (an error message). In the new Xerox Lisp error system, all errors are handled by signalling an object of type **XCL:CONDITION**. The mapping from Interlisp error numbers to Xerox Lisp conditions is given below in Section 14.10.

Since one cannot in general map a condition object to an Interlisp error number, the function **ERRORN** no longer exists. The equivalent functionality exists by examining the special variable ***LAST-CONDITION***, whose value is the condition object most recently signaled.

(ERRORX ERXM) calls **CL:ERROR** after first converting **ERXM** into a condition in the following way: If **ERXM** is **NIL**, the value of ***LAST-CONDITION*** is used; if **ERXM** is an Interlisp error descriptor, it is first converted to a condition; finally, if **ERXM** is already a condition, it is passed along unchanged. **ERRORX** also sets up a proceed case for **XCL:PROCEED**, which will attempt to re-evaluate the caller of **ERRORX**, much as OK did in the old Interlisp break package.

ERROR, HELP, SHOULDNT, RESET, ERRORMESS, ERRORMESS1, and **ERRORSTRING** work as before. All output is directed to ***ERROR-OUTPUT***, initially the terminal.

ERROR! is equivalent to the new error system's **XCL:ABORT** proceed function, except that if no **ERRORSET** or **XCL:CATCH-ABORT** is found, it unwinds all the way to the top of the process.

SETERRORN converts its arguments into a condition, then sets the value of ***LAST-CONDITION*** to the result.

Section 14.8 Catching Errors

ERRORSET, ERSETQ and **NLSETQ** have been reimplemented in terms of the new error system, but their behavior is essentially the same as before. **NLSETQ** catches all errors (conditions of type **CL:ERROR** and its descendants), and sets up a proceed case for **XCL:ABORT** so that **ERROR!** will return from it. **ERSETQ** also sets up a proceed case for **XCL:ABORT**, though it does not catch errors.

One consequence of the new implementation is that there are no longer frames named **ERRORSET** on the stack; programs that explicitly searched for such frames will have to be changed.

ERRORTYPELIST is no longer supported. The equivalent functionality is provided by default handlers. Although condition handlers provide a more powerful mechanism for programmatically responding to an error condition, old **ERRORTYPELST** entries generally cannot be translated directly. Condition handlers that want to resume a computation (rather than, say, abort from a well-know stack location) generally require the cooperation of a proceed case in the signalling code; there is no easy way to provide a substitute value for the "culprit" to be re-evaluated in a general way.

One important difference between **ERRORTYPELIST** and condition handlers is their behavior with respect to **NLSETQ**. In Koto, the relevant error handler on **ERRORTYPELST** would be tried, even for errors occurring underneath an **NLSETQ**. In Lyric, the **NLSETQ** traps all errors before the default condition handlers can see the error. This means, for example, that the behavior of **(NLSETQ (OPENSTREAM --))** is now different if the **OPENSTREAM** causes a file not found error—in Koto, the system would search **DIRECTORIES** for the file; in Lyric, the **NLSETQ** returns **NIL** immediately without searching, since the default handler for **XCL:FILE-NOT-FOUND** is not invoked.

Section 14.9 Changing and Restoring System State

The special forms **RESETLST, RESETSAVE, RESETVAR, RESETVARS** and **RESETFORM** still exist, but are implemented by a new mechanism that also supports Common Lisp's **CL:UNWIND-PROTECT**. Common Lisp's **CL:THROW** and (in most cases) Interlisp's **RETFROM** and related control transfer constructs cause the cleanup forms of both **CL:UNWIND-PROTECT** and **RESETLST** (etc) to be performed. This is discussed in more detail in the notes for Chapter 11, the stack.

Section 14.10 Error List

Most of the Interlisp errors are mapped into condition types in Xerox Lisp. Some are no longer supported. Following is the list of error type mappings. The first name is the condition type that the error descriptor turns into. If there is a second name, it is the slot whose value is set to **CADR** of the error descriptor. Any additional pairs of items are the values of other slots set by the mapping. Attempting to use an unsupported error type number will result in a simple error to that effect.

0 Obsolete

1 Obsolete

2 **STACK-OVERFLOW**

3 **ILLEGAL-RETURN**

4 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'LIST**

5 **XCL:SIMPLE-DEVICE-ERROR** *MESSAGE*

6 **XCL:ATTEMPT-TO-CHANGE-CONSTANT**

7 **XCL:ATTEMPT-TO-RPLAC-NIL** *MESSAGE*

8 **ILLEGAL-GO TAG**

9 **XCL:FILE-WONT-OPEN PATHNAME**

10 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'CL:NUMBER**

11 **XCL:SYMBOL-NAME-TOO-LONG**

12 **XCL:SYMBOL-HT-FULL**

13 **XCL:STREAM-NOT-OPEN** *STREAM*

14 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'CL:SYMBOL**

15 Obsolete

16 **END-OF-FILE** *STREAM*

17 **INTERLISP-ERROR** *MESSAGE*

18 Not supported (control-B interrupt)

19 **ILLEGAL-STACK-ARG** *ARG*

20 Obsolete

21 **XCL:ARRAY-SPACE-FULL**

22 **XCL:FS-RESOURCES-EXCEEDED**

23 **XCL:FILE-NOT-FOUND** *PATHNAME*

24 Obsolete

25 **INVALID-ARGUMENT-LIST** *ARGUMENT*

26 **XCL:HASH-TABLE-FULL** *TABLE*

27 **INVALID-ARGUMENT-LIST** *ARGUMENT*

28 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'ARRAYP**

29 Obsolete

30 **STACK-POINTER-RELEASED** *NAME*

31 **XCL:STORAGE-EXHAUSTED**

32 Not supported (attempt to use item of incorrect type)

33 Not supported (illegal data type number)

34 **XCL:DATA-TYPES-EXHAUSTED**

35 **XCL:ATTEMPT-TO-CHANGE-CONSTANT**

36 Obsolete

37 Obsolete

38 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'READTABLEP**

39 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'TERMTABLEP**

40 Obsolete

41 **XCL:FS-PROTECTION-VIOLATION**

42 **XCL:INVALID-PATHNAME** *PATHNAME*

43 Not supported (user break)

44 **UNBOUND-VARIABLE** *NAME*

45 **UNDEFINED-CAR-OF-FORM** *FUNCTION*

46 **UNDEFINED-FUNCTION-IN-APPLY**

47 **XCL:CONTROL-E-INTERRUPT**

48 **XCL:FLOATING-UNDERFLOW**

49 **XCL:FLOATING-OVERFLOW**

50 Not supported (integer overflow)

51 **XCL:SIMPLE-TYPE-ERROR** *CULPRIT* **:EXPECTED-TYPE 'CL:HASH-TABLE**

52 **TOO-MANY-ARGUMENTS** *CALLEE* **:MAXIMUM CL:CALL-ARGUMENTS-LIMIT**

Note that there are many other condition types in Xerox Lisp; see the error system documentation in the *Xerox Common Lisp Implementation Notes* for details.

Chapter 15 Breaking Functions and Debugging

The Lyric release of Xerox Lisp contains a completely new implementation of the breaking, tracing and advising facilities described in this chapter of the *Interlisp-D Reference Manual*. As a result, while the overall behavior of the functions defined in this chapter has not changed, many details of the underlying implementation have. The standard uses of BREAK, TRACE, and ADVISE are unchanged, from the user's point of view, but the internals of the implementation are quite different.

For complete documentation on the new implementation of breaking, tracing and advising, see the *Xerox Common Lisp Implementation Notes*, Section 25.3.

In particular, you should note the following differences:

- The variable **BRKINFOLST** no longer exists and the format of the value of the variable **BROKENFNS** has changed. In addition, the **BRKINFO** property is no longer used.
- **BREAK** and **TRACE** no longer work on CLISP words.
- The **BREAKIN** and **UNBREAKIN** functions no longer exist. No comparable facility exists in Xerox Lisp. The user can manually insert calls to the Common Lisp function **CL:BREAK** in order to create a breakpoint at that point in the function.

Please note the following additional changes to breaking functions:

Section 15.1 Breaking Functions and Debugging

(BREAK0 *FN WHEN COMS* — —**)** [Function]

The function **BREAK0** now works when applied to an undefined function. This allows you to use the breaking facility to create "stubs" that generate a breakpoint when called. You can then examine the arguments passed and use the **RETURN** command in the debugger to return the proper result(s).

The "break commands" facility (the *COMS* argument) is no longer supported. **BREAK0** now signals an error when supplied with a non-**NIL** third argument. If you need finer control over the functioning of breakpoints you are directed to the **ADVISE** facility; it offers complete control of how and when the given function is evaluated.

Passing a non-atomic argument in the form (*FN1* **IN** *FN2*) as the first argument to **BREAK0** still has the effect of creating a breakpoint wherever *FN2* calls *FN1*. However, it no longer creates a function named *FN1*-**IN**-*FN2* to do so. In addition, the

format of the value of the **NAMESCHANGED** property has changed and the **ALIAS** property is no longer used.

(TRACE *X***)** [Function]

TRACE is no longer a special case of **BREAK**, though the functions **UNBREAK** and **REBREAK** continue to work on traced functions.

In addition, the function **TRACE** no longer calls **BREAK0** in order to do its job. Also, non-atomic arguments to **TRACE** no longer specify forms the user wishes to see in the tracing output.

(UNBREAK *X***)** [Function]

The function **UNBREAK** is no longer implemented in terms of **UNBREAK0**, although that function continues to exist.

Section 15.2 Advising

The implementation of advising has been completely reworked. While the semantics implied by the code shown in Section 15.2.1 of the *Interlisp-D Reference Manual* is still supported, the details are quite different. In particular, it is now possible to advise functions that return multiple values and for **AFTER**-style advice to access those values. Also, all advice is now compiled, rather than interpreted. The advising facility no longer makes use of the special forms **ADV-PROG**, **ADV-RETURN**, and **ADV-SETQ**.

You should also note the following changes to the advise facility:

- The editing of advice has changed slightly. In previous releases, the advice and original function-body were edited simultaneously. In Lyric, they can only be edited separately. When you finish editing the advice for a function, that function is automatically re-advised using the new advice.
- The variable **ADVINFOLST** no longer exists and the format of the value of the variable **ADVISEDFNS** has changed. In addition, the properties **ADVICE** and **READVICE** are no longer used, except in the handling of advice saved on files from previous releases. Advice saved in Lyric does not use the **READVICE** property.
- The function **ADVISEDUMP** no longer exists.
- Advice saved on files in previous releases can, in general, be loaded into the Lyric system compatibly. A known exception is the case in which a list of the form **(***FN1* **IN** *FN2***)** was given to the **ADVICE** or **ADVISE** file package commands. When **READVISE** is called on such a name, the old-style advice, on the **READVICE** property of the symbol *FN1*-**IN**-*FN2*, will not be found. This will eventually lead to an **XCL:ATTEMPT-TO-RPLAC-NIL** error. The user should evaluate the form
  (RETFROM 'READVISE1)
  in the debugger to proceed from the error and later evaluate
  (READVISE *FN1*-**IN**-*FN2***)**
  by hand to install the advice.

- The **ADVICE** and **ADVISE** File Manager commands now accept three kinds of arguments:
  a symbol, naming an advised function,
  a list in the form (*FN1* **:IN** *FN2*), and
  a symbol of the form *FN1*-**IN**-*FN2*.
  Arguments of the form (*FN1* **IN** *FN2*) are not acceptable any longer. Arguments of the form *FN1*-**IN**-*FN2* should be converted into the equivalent form (*FN1* **:IN** *FN2*).

(**ADVISE** *WHO WHEN WHERE WHAT*) [Function]

In the Lyric release of Xerox Lisp, **ADVISE** has some changes in the way arguments are treated and the possible values for those arguments. Most notably:

- In earlier releases, you could call **ADVISE** with only one argument, the name of a function. In this case, **ADVISE** "set up" the named function for advising, but installed no advice. This usage is no longer supported.
- Previously, an undocumented value of **BIND** was accepted for the *WHEN* argument to **ADVISE**. This kind of advice is no longer supported. It can be adequately simulated using **AROUND** advice.

In addition, advising Common Lisp functions works somewhat differently with respect to a function's arguments. The arguments are not available by name. Instead, the variable **XCL:ARGLIST** is bound to a list of the values passed to the function and may be changed to affect what will be passed on.

As with the breaking facility (see above), **ADVISE** no longer creates a function named *FN1*-**IN**-*FN2* as a part of advising (*FN1* **IN** *FN2*).

Chapter 16 List Structure Editor

The list structure editor, DEdit, is not part of the Xerox Lisp environment. It is now a Lisp Library Module. Chapter 16 has been renamed Structure Editor.

SEdit, the new Lisp editor, replaces DEdit in the Lyric release. The description of SEdit may be found in Appendix B of this volume. The commands used to invoke both SEdit and DEdit are the same.

Following is a description of the interface to the Lisp editor.

Switching Between Editors

If you have both SEdit and DEdit loaded, you switch between them by calling: (**EDITMODE** *'EDITORNAME*) where *EDITORNAME* is one of the symbols SEdit or DEdit.

Starting a Lisp Editor

(ED *NAME &OPTIONAL OPTIONS***)** [Function]

Calls the currently active Lisp editor. SEdit is the default Lisp editor. The same symbol, **ED**, is used in both the IL and CL packages.

NAME is the name of any File Manager object.

OPTIONS is either a single symbol or a list of symbols, each of which is either a File Manager type or one or more of the keywords **:DISPLAY**, **:DONTWAIT**, **:CLOSE-ON-COMPLETION**, or **:NEW**. If exactly one File Manager type is given, **ED** tries to edit that type of definition for *NAME*. If more than one type is given in *OPTIONS*, **ED** will determine for which of them *NAME* has a definition. If a definition exists for more than one of the types, **ED** gives you a choice of which one to edit. If no File Manager types are given, **ED** treats *OPTIONS* as though a list of all of the existing types had been given; thus you are given a choice of all of the existing definitions of *NAME*.

The variable **FILEPKGTYPES** contains a complete list of the currently-known manager types.

If the keyword **:DISPLAY** is included in *OPTIONS*, **ED** uses menus for any prompting (e.g., to choose one of several possible definitions to edit). If **:DISPLAY** is not included, **ED** prints its queries to and reads the user's replies from ***QUERY-IO*** (usually the Exec in which you are typing). Thus all of the following are correct ways to call the editor:

```
(ED 'NAME :DISPLAY)
(ED 'NAME 'FUNCTIONS)
(ED 'NAME '(:DISPLAY))
(ED 'NAME '(FUNCTIONS :DISPLAY))
(ED 'NAME '(FUNCTIONS VARIABLES :DISPLAY))
```

The other keywords are interpreted as follows:

:DONTWAIT

Lets the edit interface return right away, rather than waiting for the edit to be complete. **DF**, **DV**, **DC**, and **DP** specify this option now, so editing from the exec will not cause the exec to wait.

:NEW

Lets you install a new definition for the name to be edited. You will be asked what type of dummy definition you wish to install based on which file manager types were included in *OPTIONS*.

:CLOSE-ON-COMPLETION

Tells the editor that it must close the editor window after the first completion. So in SEdit, CONTROL-X will close the window; shrinking the window is not allowed. Editor windows opened by the exec command **FIX** specify this option.

If *NAME* does not have a definition of any of the given types, **ED** can create a dummy definition of any of those types. You will be asked to select what type you wish to install. New kinds of

dummy definitions can be added to the system through the use of the **:PROTOTYPE** option to **XCL:DEFDEFINER**.

Mapping the Old Edit Interface to ED

The old functions for starting the Lisp editor (**DF**, **DV**, **DP**, and **DC**) have been reimplemented so that they work with Common Lisp. The old edit commands map to the new editor function (ED) as follows:

```
DF NAME ⇒ (ED 'NAME '(FUNCTIONS FNS :DONTWAIT))
DV NAME ⇒ (ED 'NAME '(VARIABLES VARS :DONTWAIT))
DP NAME ⇒ (ED 'NAME '(PROPERTY-LIST :DONTWAIT))
DP NAME MYPROP ⇒ (ED '(NAME MYPROP) '(PROPS :DONTWAIT))
DC NAME ⇒ (ED 'NAME '(FILES :DONTWAIT))
```

Thus, for example, when **DF** is invoked it looks first for Common Lisp FUNCTIONS and then for Interlisp FNS. **DV**, **DP** and **DC** operate in an analogous fashion.

Section 16.18 Editor Functions

(II:16.74)

The function **FINDCALLERS** has the following limitations in Xerox Lisp:

1. **FINDCALLERS** only identifies by name the occurrences inside of Interlisp FNS, not Common Lisp FUNCTIONS.

2. Because **FINDCALLERS** uses a textual search, it may report more occurrences of the specified *ATOMS* than there actually are, if the file contains symbols by the same name in another package, or symbols with the same pname but different alphabetic case. **EDITCALLERS** still edits only the actual occurrences, since it reads the functions and operates on the real Lisp structure, not its printed representation.

Chapter 17 File Package

The Interlisp-D File Package has been renamed the File Manager. Its operation is unchanged; however, it has been extended to manipulate, load and save Common Lisp functions, variables, etc. It also allows specification of the reader environment (package and readtable) to use when writing and reading a file, solving the problem of compatibility between old and new (Common Lisp) syntax.

Note that although source files from earlier releases can be loaded into Lyric, files produced by the File Manager in the Lyric release cannot be loaded into previous releases. This is true for several reasons, the most important being that previous releases did not have packages, so symbols cannot be read back consistently.

The new File Manager includes several new types to deal with the various definition forms supported in Xerox Common Lisp. The following table associates each new type with the forms that produce definitions of that type:

FUNCTIONS	**CL:DEFUN, CL:DEFMACRO, CL:DEFINE-MODIFY-MACRO, XCL:DEFINLINE, XCL:DEFDEFINER. XCL:DEFINE-PROCEED-FUNCTION**
VARIABLES	**CL:DEFCONSTANT, CL:DEFVAR, CL:DEFPARAMETER, XCL:DEFGLOBALVAR, XCL:DEFGLOBALPARAMETER**
STRUCTURES	**CL:DEFSTRUCT, XCL:DEFINE-CONDITION**
TYPES	**CL:DEFTYPE**
SETFS	**CL:DEFSETF, CL:DEFINE-SETF-METHOD**
DEFINE-TYPES	**XCL:DEF-DEFINE-TYPE**
OPTIMIZERS	**XCL:DEFOPTIMIZER**
COMMANDS	**XCL:DEFCOMMAND**

Note that the types listed above, as well as all the old File Manager types, are symbols in the INTERLISP package. In addition, the "filecoms" variable of a file and its rootname are also both in the INTERLISP package. You should be careful when typing to a Common Lisp exec to qualify all such symbols with the prefix **IL:**. E.g.,

3>**(setq il:foocoms '((il:functions bar) (il:prop il:filetype il:foo)))**

to indicate you want the function BAR (in the current package) to live on a file with rootname FOO, and also that FOO's FILETYPE property should be saved.

Reader Environments and the File Manager

(II:17.1)

In order for **READ** to correctly read back the same expression that **PRINT** printed, it is necessary that both operations be performed in the same reader environment, i.e., the collection of parameters that affect the way the reader interprets the characters appearing on the input stream. In previous releases of Interlisp there was, for all practical purposes, a single such environment, defined entirely by the readtable *FILERDTBL*. In the Lyric release of Xerox Lisp there are two significantly different readtables in which to read (Common Lisp and Interlisp). In addition, there are more parameters than just the readtable that can potentially affect **READ**: the current package and the read base (the bindings of ***PACKAGE*** and ***READ-BASE***).

To handle this diversity, a new type of object is introduced, the **READER-ENVIRONMENT**, consisting of a readtable, a package, and a read/print base. Every file produced by the File Manager has a header at the beginning specifying the reader environment for that file. **MAKEFILE** and the compiler produce this header, while **LOAD**, **LOADFNS**, and other file-reading functions read the header in order to set their reading environment correctly. Files

written in older releases of Lisp lack this header and are interpreted as having been written in the environment consisting of the readtable *FILERDTBL* and the package INTERLISP. Thus, you need take no special action to be able to load Koto source files into Lyric; characters that are "special" in Common Lisp, such as colon, semi-colon and hash, are interpreted as the "ordinary" characters they were in Koto.

The File Manager's reader environments are specified as a property list of alternating keywords and values of the form (**:READTABLE** *readtable* **:PACKAGE** *package* **:BASE** *base*). The **:BASE** pair is optional and defaults to 10. The values for *readtable* and *package* should either be strings naming a readtable and package, or expressions that can be evaluated to produce a readtable and package. In the former case, the readtable or package *must* be one that already exists in a virgin Lisp sysout (or at least in any Lisp image in which you might attempt *any* operation that reads the file). If an expression is used, care should be exercised that the expression can be evaluated in an environment where no packages or readtables, other than the documented ones, are presumed to exist. For hints and guidelines on writing the *package* expression for files that create or use their own private packages, please see Chapter 11 of the *Xerox Common Lisp Implementation Notes*.

When **MAKEFILE** is writing a source file, it uses the following algorithm to determine the reading environment for the new file:

1. If the root name for the file has the property **MAKEFILE-ENVIRONMENT**, the property's value is used. It should be in the form described above. Note that if you want the file always to be written in this environment, you should save the **MAKEFILE-ENVIRONMENT** property itself on the file, using a (**PROP MAKEFILE-ENVIRONMENT** *file*) command in the filecoms.

2. If a previous version of the file exists, **MAKEFILE** uses the previous version's environment. **MAKEFILE** does this even when given option **NEW** or the previous version is no longer accessible, assuming it still has the previous version's environment in its cache. If the previous version was written in an older release, and hence has no explicit reader environment, **MAKEFILE** uses the environment (**:READTABLE** "INTERLISP" **:PACKAGE** "INTERLISP" **:BASE** 10).

3. If no previous version exists (this is a new file), **MAKEFILE** uses the value of ***DEFAULT-MAKEFILE-ENVIRONMENT***, initially (**:READTABLE** "XCL" **:PACKAGE** "INTERLISP" **:BASE** 10).

Note that changing the value of ***DEFAULT-MAKEFILE-ENVIRONMENT*** only affects new files. If you decide you don't like the environment in which an existing file is written, you must give the file a **MAKEFILE-ENVIRONMENT** property to override any prior default.

Since the XCL readtable is case-insensitive, you should avoid using it for files that contain many mixed-case symbols or old-style Interlisp comments, as these will be printed with many

escape delimiters. This is why the default for reprinted Koto sources is the INTERLISP readtable.

The readtable named LISP (the pure Common Lisp readtable) should ordinarily not be used as part of a **MAKEFILE** environment. It exists solely for the use of "pure" Common Lisp (as in the CL Exec), and thus has no provision for font escapes (inserted by the Xerox Lisp prettyprinter) to be treated as whitespace. Most users will want to use either XCL or INTERLISP as the readtable for files.

If the environment for the new version of the file differs from that of the previous version, **MAKEFILE** copies unchanged FNS definitions by actually reading from the old file, rather than just copying characters as it otherwise would. Similarly, when **RECOMPILE** or **BRECOMPILE** attempt to recompile a file for which the previous compiled version's reader environment is different, they must compile afresh all the functions on the file, i.e., they behave like **TCOMPL** or **BCOMPL**.

Modifying Standard Readtables

In the past, programmers have been periodically tempted to change standard readtables, such as **T** and **FILERDTBL**, typically by adding macros to read certain objects in a convenient way. For example, the PQUOTE LispUsers module defined single quote as a macro in **FILERDTBL**. Unfortunately, changing a standard readtable means that unless you are very careful, you cannot read other users' files that were not written with your change, and they cannot read your files without obtaining your macro. Furthermore, the effects are often subtle. Rather than breaking, the system merely reads the file incorrectly. For example, reading a file written with PQUOTE in an environment lacking PQUOTE produces many symbols with a single quote packed on the front.

This confusion can be avoided with MAKEFILE reader environments. To add your own special macro:

1. Copy some standard readtable; e.g., (COPYRDTBL "INTERLISP").
2. Give it a distinguished name of its own, by using (READTABLEPROP *rdtbl* 'NAME "*yourname*").
3. Make your change in the copied readtable.
4. Use your new private readtable to write your files: use its name ("*yourname*") in the **MAKEFILE-ENVIRONMENT** property of selected files and/or change ***DEFAULT-MAKEFILE-ENVIRONMENT*** to affect all your new files.
5. Make sure to save your new readtable. It is usually most convenient to include the code to create it (steps 1-3) in your system initialization, but you could even write a self-contained expression to use in a single file's **MAKEFILE-ENVIRONMENT** property.

With this strategy, your system will read all files in the proper environment—your own files with your private readtable and other users' files in their environments, including the standard environments, which you have carefully avoided polluting. If another user tries to load one of your files into an environment that doesn't know about your private readtable, **LOAD** will give an error immediately (readtable not found), rather than loading the file quietly but incorrectly.

Programmer's Interface to Reader Environments

The following function and macro are available for programmers to use. Note that reader environments only control the parameters that determine read/print consistency. There are other parameters, such as ***PRINT-CASE***, that affect the appearance of the output without affecting its ability to be read. Thus, reader environments are not sufficient to handle problems of, for example, repainting expressions on the display in exactly the same total environment in which they were first written.

(MAKE-READER-ENVIRONMENT *PACKAGE READTABLE BASE***)** [Function]

Creates a **READER-ENVIRONMENT** object with the indicated components. The arguments must be valid values for the variables ***PACKAGE***, ***READTABLE*** and ***PRINT-BASE***; names are not sufficient. If any of the arguments is **NIL**, the current value of the corresponding variable is used. Thus **(MAKE-READER-ENVIRONMENT)** returns an object that captures the current environment.

(WITH-READER-ENVIRONMENT *ENVIRONMENT . FORMS***)** [Macro]

Evaluates each of the *FORMS* with ***PACKAGE***, ***READTABLE***, ***PRINT-BASE*** and ***READ-BASE*** bound to the values in the *ENVIRONMENT* object. Both ***PRINT-BASE*** and ***READ-BASE*** are bound to the single *BASE* value in the environment.

(GET-ENVIRONMENT-AND-FILEMAP *STREAM DONTCACHE***)** [Function]

Parses the header of a file produced by the File Manager and returns up to four values:

1. The reader environment in which the file was written;
2. The file's "filemap", used to locate functions on the file;
3. The file position where the FILECREATED expression starts; and
4. A value used internally by the File Manager.

STREAM can be a full file name, in which case this function returns NIL unless the information was previously cached. Otherwise, *STREAM* is a stream open for input on the file. It must be randomly accessible (unless information is available from the cache). If the file is in Common Lisp format (it begins with a comment), then value 1 is the default Common Lisp reader environment (readtable LISP, package USER) and the other values are NIL. Otherwise, if the file is not in File Manager format, values 1 and 2 are NIL, 3 is zero.

If *DONTCACHE* is true, the function does not cache any information it learns about File Manager files; otherwise, the information is cached to speed up future inquiries.

Section 17.1 Loading Files

(II:17.5)

Integration of Interlisp and Common Lisp LOAD functions

There are four kinds of files that can be loaded in Xerox Lisp:

1. Interlisp and Common Lisp source files produced by the File Manager using, for example, the **MAKEFILE** function.
2. Standard Common Lisp source files produced with a text editor either in Xerox Lisp or from some other Common Lisp implementation.
3. DFASL files of compiled code, produced by the new XCL Compiler, **CL:COMPILE-FILE** (extension DFASL)
4. LCOM files of compiled code, produced by the old Interlisp Compiler (**BCOMPL, TCOMPL**),

Types 1 and 4 were the only kind of files that you could load in Koto; types 2 and 3 are new with Lyric. Both **IL:LOAD** and **CL:LOAD** are capable of loading all four kinds of files. However, they use the following rules to make the types of files unambiguous so that they can be loaded in the correct reader environment.

- If the file begins with an open parenthesis (possibly after whitespace and font switch characters), it is assumed to be of type 1 or 4: files produced by the File Manager. The first expression on the file (at least) is assumed to be written in the old **FILERDTBL** environment; for new Lyric files this expression defines the reader environment for the remainder of the file. See the section, Reader Environments and File Manager for details.
- If the file begins with the special FASL signature byte (octal 221), it is assumed to be a compiled file in FASL format, and is processed by the FASL loader. The FASL loader ignores the *LDFLG* argument to **IL:LOAD**, treating all files as though *LDFLG* were **SYSLOAD** (redefinition occurs, is not undoable, and no File Manager information is saved).
- If the file begins with a semicolon, it is assumed to be a pure Common Lisp file. The expressions on the file are read with the standard Common Lisp readtable and in package USER (unless a package argument was given to **LOAD**; see below).
- If the file begins with any other character, **LOAD** doesn't know what to do. Currently, it treats the file as a pure Common Lisp file (as if it started with a comment).

Thus, if you prepare Common Lisp text files you should be sure to begin them with a comment so that **LOAD** can tell the file is in Common Lisp syntax.

The function **CL:LOAD** accepts an additional keyword **:PACKAGE**, whose value must be a package object; the function **IL:LOAD** similarly has an optional fourth argument *PACKAGE*. If a package argument is given, then **LOAD** reads Common Lisp text files (type 2 above) with ***PACKAGE*** bound to the specified package. In the case of File Manager files (types 1 and 4), the package argument overrides the package specified in the file's reader environment.

(II:17.6-17.8)

The Interlisp functions **LOADFNS**, **LOADFROM**, **LOADVARS** and **LOADCOMP** do not work on FASL files. They do still work on files produced by the old compiler (extension LCOM).

(II:17.9)

FILESLOAD (also used by the File Manager's **FILES** command) now searches for compiled files by looking for a file by the specified name whose extension is in the list ***COMPILED-EXTENSIONS***. The default value for ***COMPILED-EXTENSIONS*** the Lyric release is (DFASL LCOM). It searches the list of extensions in order for each directory on the search path. This means that FASL files are loaded in preference to old-style compiled files.

Section 17.2 Storing Files

The Lyric release contains two different compilers, the Interlisp Compiler that was present in Koto and previous releases, and the new XCL Compiler (see the next section, Chapter 18 Compiler). With more than one compiler available, the question arises as to which compiler will be used by the functions **CLEANUP** and **MAKEFILE**. The default behavior of these functions in Lyric is to always use the new XCL Compiler. This default can be changed, either on a file-by-file basis or system-wide. Most users, however, will have no need to change the default.

When the *C* or *RC* option has been given to **MAKEFILE**, the system first looks for the value of the **FILETYPE** property on the symbol naming the file. For example, for the file "{DSK}<LISPFILES>MYFILE", the property list of the symbol **MYFILE** would be examined.

The **FILETYPE** property should be either a symbol from the list below or a list containing one of those symbols. The following symbols are allowed and have the given meanings:

:TCOMPL Compile this file by calling either **TCOMPL** or **RECOMPILE**, depending upon which of the C or RC options was passed to **MAKEFILE**.

:BCOMPL Compile this file by calling either **BCOMPL** or **BRECOMPILE**, depending upon which of the C or RC options was passed to **MAKEFILE**. This is equivalent to the Koto behavior.

:COMPILE-FILE Compile this file by calling **CL:COMPILE-FILE**, regardless of which option was passed to **MAKEFILE**.

If no **FILETYPE** property is found, then the function whose name is the value of the variable ***DEFAULT-CLEANUP-COMPILER*** s

used. The only legal values for this variable are **TCOMPL**, **BCOMPL**, and **CL:COMPILE-FILE**. Initially, ***DEFAULT-CLEANUP-COMPILER*** is set to **CL:COMPILE-FILE**.

If you choose to set the **FILETYPE** property of file name, you should take care that the filecoms for that file saves the value of that property on the file. This will ensure that the same compiler will be used every time the file is loaded. To save the value of the property, you should include a line in the coms like the following:

```
(PROP FILETYPE MYFILE)
```

where MYFILE is the symbol naming your file.

Section 17.8.2 Defining New File Manager Types

(II:17.30)

The File Manager has been extended to allow File Manager types that accept any Lisp object as a name. A consequence of this is that any user-defined type's **HASDEF** function should be prepared to accept objects other than symbols as the *NAME* argument. Names are compared using **EQUAL**.

Definers: A New Facility for Extending the File Manager

The Definer facility is provided to make the process of adding a certain common kind of File Manager type easy. All of the new File Manager types in the Lyric release (including **FUNCTIONS**, **VARIABLES**, **STRUCTURES**, etc.) and almost all of the new defining macros (including **CL:DEFUN**, **CL:DEFPARAMETER**, **CL:DEFSTRUCT**, etc.) were themselves created using the Definer facility.

In previous releases, adding new types and commands to the File Manager involved deeply understanding the way in which it worked and defining a number of functions to carry out certain operations on the new type/command. Further, making functions and macros save away definitions of the new type was similarly subtle and generally difficult or complicated to do. With the addition of Common Lisp, it was realized that a large number of new types and commands would be added, all needing essentially the same implementation of the various operations. In addition, many new defining macros were to be added and all of them needed to save definitions.

As an explanation of the Definer facility, we will describe how **VARIABLES** and **CL:DEFPARAMETER** could be added into the system, if they were not already there.

First, a little background about our example. The macro **CL:DEFPARAMETER** is used in Common Lisp to globally declare a given variable to be special and to give it an initial value. (For the purposes of this example, we will ignore the documentation-string given to real **CL:DEFPARAMETER** forms.) The value of a call to the macro should be the name of the variable being defined. An acceptable definition of this macro might appear as follows:

```
(DEFMACRO CL:DEFPARAMETER (SYMBOL EXPRESSION)
   `(PROGN
      (CL:PROCLAIM '(CL:SPECIAL ,SYMBOL))
      (SETQ ,SYMBOL ,EXPRESSION)
      ',SYMBOL))
```

There are some problems with using such a simple definition in the Xerox Lisp environment, however. For example, if a call to this macro were typed to the Exec, the File Manager would not be told to notice it. Thus, there would be no convenient way to remember to add the form to the filecoms of some file and thus to save it away. Also, note that the macro does not pay attention to the **DFNFLG** variable; thus, loading a file containing a **CL:DEFPARAMETER** form would always set the variable to the value of the initial expression, even when **DFNFLG** was set to **ALLPROP**. This could make editing code using this variable difficult.

We will now proceed to fix these problems by getting the Definer facility involved. There are two steps involved in using Definers:

- Unless one of the currently-existing File Manager types is appropriate for definitions using the new macro, a new type must be created. The macro **XCL:DEF-DEFINE-TYPE** is used for this purpose.
- The macro must be defined in such a way that the File Manager can tell that it should notice and save uses of the macro and under which File Manager type the uses should be saved. The macro **XCL:DEFDEFINER** is used for this purpose.

Since we are pretending for the example that the File Manager type **VARIABLES** is not defined, we decide that definitions using **CL:DEFPARAMETER** should not be given any of the already-existing types. We must define a type, therefore, and we decide to call it **VARIABLES**. The following **XCL:DEF-DEFINE-TYPE** form will do the trick:

```
(XCL:DEF-DEFINE-TYPE VARIABLES "Common Lisp
variables")
```

The first argument to **XCL:DEF-DEFINE-TYPE** is the name for the new type. The second argument is a descriptive string, to be used when printing out messages about the type.

With the new type thus created, we can now use **XCL:DEFDEFINER** to redefine the macro. Simply changing the word **DEFMACRO** into **XCL:DEFDEFINER** and adding an argument specifying the new type suffices to change our earlier definition into a use of the Definer facility:

```
(XCL:DEFDEFINER CL:DEFPARAMETER VARIABLES
                                (SYMBOL EXPRESSION)
   `(PROGN
      (CL:PROCLAIM '(CL:SPECIAL ,SYMBOL))
      (SETQ ,SYMBOL ,EXPRESSION)
      ',SYMBOL))
```

(In fact, we could also remove the final **',SYMBOL**; **XCL:DEFDEFINER** automatically arranges for the new macro to

return the name of the new definition.) Now, if we were to type the form

```
(CL:DEFPARAMETER *FOO* 17)
```

into the Exec and then call the function **FILES?**, we would be presented with something like the following:

```
24> (FILES?)
the Common Lisp variables: *FOO*
...to be dumped.  want to say where the above
go?
```

As with other File Manager types, our definitions are being kept track of. If we answer Yes to the above question and specify a file in which to save the definition, a command like the following will be added to the filecoms:

```
(VARIABLES *FOO*)
```

Actually, the output from **FILES?** as shown above is not quite accurate. In reality, we would also be asked about the following:

```
the Common Lisp functions/macros:
CL:DEFPARAMETER
the Definition types: VARIABLES
```

The File Manager is also watching for new types and new Definers being created and will let us save those definitions as well. These would be listed in the filecoms as follows:

```
(DEFINE-TYPES VARIABLES)
(FUNCTIONS CL:DEFPARAMETER)
```

All of these definitions are full-fledged File Manager citizens. The functions **GETDEF**, **HASDEF**, **PUTDEF**, **DELDEF**, etc. all work with the new type. We can edit the definition of ***FOO*** above simply by specifying the type to the **ED** function:

```
(ED '*FOO* 'VARIABLES)
```

When we exit the editor, the new definition will be saved and, unless **DFNFLG** is set to **PROP** or **ALLPROP**, evaluated.

It is now time to fully describe the macros **XCL:DEF-DEFINE-TYPE** and **XCL:DEFDEFINER**.

XCL:DEF-DEFINE-TYPE *NAME DESCRIPTION* &KEY :UNDEFINER [*Macro*]

Creates a new File Manager type and command with the given *NAME*. The string *DESCRIPTION* will be used to describe the type in printed messages. The new type implements **PUTDEF** operations by evaluating the definition form, **GETDEF** and **HASDEF** by looking up the given name in an internal hash-table, using **EQUAL** as the equality test on names, and **DELDEF** by removing any named definition from the hash-table. If the **:UNDEFINER** argument is provided, it should be the name of a function to be called with the *NAME* argument to any **DELDEF** operations on this type. The **:UNDEFINER** function can perform any other operations necessary to completely delete a definition.

XCL:DEF-DEFINE-TYPE forms are File Manager definitions of type **DEFINE-TYPES**.

As an example of the full use of **XCL:DEF-DEFINE-TYPE**, here is the complete definition of the type VARIABLES as it exists in the Lyric release:

```
(XCL:DEF-DEFINE-TYPE VARIABLES "Common Lisp variables"
                     :UNDEFINER UNDOABLY-MAKUNBOUND)
```

The function **UNDOABLY-MAKUNBOUND** is described in Appendix D of these Release Notes.

XCL:DEFDEFINER {*NAME* | (*NAME* {*OPTION*}*)} *TYPE ARG-LIST* &BODY *BODY* [*Macro*]

Creates a macro named *NAME*, calls to which are seen as File Manager definitions of type *TYPE*. *TYPE* must be a File Manager type previously defined using **XCL:DEF-DEFINE-TYPE**. *ARG-LIST* and *BODY* are precisely as in **DEFMACRO**. A macro defined using **XCL:DEFDEFINER** differs from one defined using **DEFMACRO** in the following ways:

- *BODY* will be evaluated if and only if the value of **DFNFLG** is not one of **PROP** or **ALLPROP**.
- The form returned by *BODY* will be evaluated in a context in which the File Manager has been temporarily disabled. This allows Definers to expand into other Definers without the subordinate ones being noticed by the File Manager.
- Calls to Definers return the name of the new definition (as, for example, **CL:DEFUN** and **CL:DEFPARAMETER** are defined to do).
- Calls to Definers are noticed and remembered by the File Manager, saved as a definition of type *TYPE*.
- SEdit- and Interlisp-style comment forms (those with a CAR of IL:*) are stripped from the macro call before it is passed to *BODY*. (This comment-removal is partially controlled by the value of the variable ***REMOVE-INTERLISP-COMMENTS***, described below.)

The following *OPTION*s are allowed:

(:UNDEFINER *FN*)

If **DELDEF** is called on a name whose definition is a call to this Definer, *FN* will be called with one argument, the name of the definition. This option allows for Definer-specific actions to be taken at **DELDEF** time. This is useful when more than one Definer exists for a given type. *FN* should be a form acceptable as the argument to the **FUNCTION** special form.

(:NAME *NAME-FN*)

By default, the Definer facility assumes that the first argument to any macro defined using **XCL:DEFDEFINER** will be the name under which the definition should be saved. This assumption holds true for almost all Common Lisp defining macros, including **CL:DEFUN**, **CL:DEFMACRO**, **CL:DEFPARAMETER** and **CL:DEFVAR**. It doesn't work, however, for a few other forms, such as **CL:DEFSTRUCT** and **XCL:DEFDEFINER** itself. When defining a macro for which that assumption is false, the **:NAME** option should be used. *NAME-FN* should be a function of one

argument, a call to the Definer. It should return the Lisp object naming the given definition (most commonly a symbol, but any Lisp object is permissible). For example, the **:NAME** option in the definitions of **CL:DEFSTRUCT** and **XCL:DEFDEFINER** is as follows:

```
(:NAME (LAMBDA (FORM)
          (LET ((NAME (CADR FORM)))
             (COND ((LITATOM NAME)
                    NAME)
                   (T  (CAR NAME))))))
```

NAME-FN should be a form acceptable as the argument to the **FUNCTION** special form (i.e., a symbol naming a function or a LAMBDA-form).

(:PROTOTYPE *DEFN-FN*)

When the editor function **ED** is passed a name with no definitions, the user is offered a choice of several ways to create a prototype definition. Those choices are specified with the **:PROTOTYPE** option to **XCL:DEFDEFINER**. *DEFN-FN* should be a function of one argument, the name to be defined using this Definer. *DEFN-FN* should return either NIL, if no definition of that name can be created with this Definer, or a form that, when evalauted, would create a definition of that name. For example, the **:PROTOTYPE** option for **CL:DEFPARAMETER** might look as follows:

```
(:PROTOTYPE (LAMBDA (NAME)
              (AND (LITATOM NAME)
                   `(CL:DEFPARAMETER ,NAME "Value"))))
```

An example using all of the features of **XCL:DEFDEFINER** is the definition of **XCL:DEFDEFINER** itself, which begins as follows:

```
(XCL:DEFDEFINER (XCL:DEFDEFINER
                 (:UNDEFINER \DELETE-DEFINER)
                 (:NAME
                    (LAMBDA (FORM)
                       (LET ((NAME (CADR FORM)))
                          (COND ((LITATOM NAME)
                                 NAME)
                                (T  (CAR NAME))))))
                 (:PROTOTYPE
                    (LAMBDA (NAME)
                       (AND (LITATOM NAME)
                            `(XCL:DEFDEFINER ,NAME "Type"
                                             ("Arg List")
                                             "Body")))))
                FUNCTIONS
                (NAME-AND-OPTIONS TYPE ARG-LIST &BODY BODY)
   ...)
```

The following variable is used in the process of removing SEdit- and Interlisp-style comments from Definer forms:

REMOVE-INTERLISP-COMMENTS [*Variable*]

Interlisp-style comments are forms whose **CAR** is the symbol **IL:***. It is possible for certain lists in Lisp code to begin with **IL:*** but not be a comment (for example, a **SELECTQ** clause). When such a list is discovered, the value of ***REMOVE-INTERLISP-COMMENTS*** is examined. If it is **T**, the list is assumed to be a comment and is removed without comment. If it is **:WARN**, a warning message is printed, saying that a possible comment was not stripped from

the code. If ***REMOVE-INTERLISP-COMMENTS*** is **NIL**, the list is not removed, but no warning is printed. This variable is initially set to **:WARN**.

Chapter 18 Compiler

The Lyric release contains two distinct Lisp compilers:

- The Interlisp Compiler, described in detail in Section 18 of the *Interlisp Reference Manual*
- The new XCL Compiler, described in the *Xerox Common Lisp Implementation Notes*.

The Interlisp Compiler provides compatibility with previous releases of Interlisp-D. It continues to work in very much the same way as it did in Koto; as before, it compiles all of the Interlisp language. The Interlisp Compiler does not, however, compile the Common Lisp language and will not be extended to do so. The Lyric release is the last release to contain the Interlisp Compiler as a component; future releases will have only the new XCL Compiler. The XCL Compiler is designed to handle both Interlisp and Common Lisp.

Several incompatible changes have been made in the compiled object code produced by the Interlisp Compiler. This means that *all user code must be recompiled in Lyric*. Code compiled in Koto or previous releases will not load into Lyric, and code compiled in Lyric wil not load into earlier releases. The filename extension for Interlisp compiled files has been changed from DCOM to LCOM in order to minimize possible confusion.

The XCL Compiler writes its output on a new kind of object file, the DFASL file. These files are quite different from the DCOM/LCOM files produced by the Interlisp Compiler. DFASL files are somewhat more compact, much faster to load and can represent a wider range of data objects than was possible in LCOMs.

Interlisp source files from Koto can be compiled using the new XCL compiler. However, some files need to be remade in Lyric before compilation: files containing bitmaps, Interlisp arrays, or the **UGLYVARS** and/or **HORRIBLEVARS** File Manager commands. To compile such a file, first **LOAD** it, then call **MAKEFILE** to write it back out. This action causes the bitmaps and other unusual objects to be written back in a format acceptable to the new compiler.

The default behavior of the File Manager's **CLEANUP** and **MAKEFILE** functions is to use the new XCL Compiler to compile files, rather than the old Interlisp Compiler. To change this behavior, see Section 17.2, Storing Files.

Note that if you call the compiler explicitly, rather than via **CLEANUP** or **MAKEFILE**, you should be careful to specify the correct compiler. The new compiler is invoked by calling

CL:COMPILE-FILE. If you inadvertantly call **BCOMPL** on a file for which **CLEANUP** has routinely been using the new XCL compiler, there are two undesirable consequences: (1) Any Common Lisp functions on the file will not be compiled (the Interlisp compiler does not recognize **CL:DEFUN**), and (2) the DFASL files produced by earlier calls on the XCL compiler will still be loaded by **FILESLOAD** in preference to the LCOM file produced by **BCOMPL**.

With this compiler, Xerox Lisp provides a facility, **XCL:DEFOPTIMIZER**, by which you can advise the compiler about efficient compilation of certain functions and macros. **XCL:DEFOPTIMIZER** works with both the old Interlisp Compiler and the new XCL Compiler. See the *Xerox Common Lisp Implementation Notes* for a description of the compiler.

Chapter 19 Masterscope

Masterscope is now a Lisp Library Module, not part of the environment.

Chapter 21 CLISP

CLISP infix forms do not work under the Common Lisp evaluator; only "clean" CLISP prefix forms are supported. You should run DWIMIFY in Koto on all other CLISP code before attempting to load it in Lyric. The remainder of this note describes the specific limitations on CLISP in Lyric.

There are two broad classes of transformations that DWIM applies to Lisp code:

1. A sort of macro expander that transforms **IF**, **FOR**, **FETCH**, etc. forms into "pure" Lisp code in well-defined ways.
2. A heuristic "corrector" that performs spelling correction and transforms CLISP infix forms such as X + Y into (PLUS X Y), sometimes having to make guesses as to whether X + Y might really have been the name of a variable.

An operational way of distinguishing the two is that DWIMIFY applied to code of type (1) makes no alterations in the code, whereas for code of type (2) it physically changes the form. Another difference is that code of type (2) must be dwimified before it can be compiled (user typically sets **DWIMIFYCOMPFLG** to **T**), whereas the compiler is able to treat code of type (1) as a special kind of macro.

Broadly speaking, code of type (2) is no longer fully supported. In particular, DWIM is invoked only when the code is encountered by the Interlisp evaluator. This means code typed to an "Old Interlisp" Executive, and code inside of an interpreted Interlisp function. Furthermore, some CLISP infix forms no

longer DWIMIFY correctly. It is likely that CLISP infix will not be supported at all in future releases.

Expressions typed to the new Executives and inside of Common Lisp functions are run by the Common Lisp evaluator (**CL:EVAL**). As far as this evaluator is concerned, DWIM does not exist, and forms beginning with "CLISP" words (**IF**, **FOR**, **FETCH**, etc) are macros. These macros perform no DWIM corrections, so all of the subforms must be correct to begin with. This is a change from past releases, where the DWIM expansion of a CLISP word form also had the side effect of transforming any CLISP infix that it might have contained. For example, the macro expansion of

```
(if X then Y+1)
```

treats Y + 1 as a variable, rather than as an addition. The correct form is

```
(if X then (PLUS Y 1)),
```

which is the way an explicit call to DWIMIFY would transform it.

If you have CLISP code from Koto you are advised to DWIMIFY the code before attempting to run or compile it in Lyric. Because of differences in the environments, not all CLISP constructs will DWIMIFY correctly in Lyric. In particular, the following do not work reliably, or at all:

1. The list-composing constructs using < and > do not DWIMIFY if the < is unpacked (an isolated symbol), because in Common Lisp, < is a perfectly valid CAR of form. On the other hand, the closing > *must* be unpacked if the last list element is quoted, since, for example, (<A 'B>) reads as (<A (QUOTE B>)).

2. Because of the conventional use of the characters * and - in Common Lisp names, those characters are only recognized as CLISP operators when they appear unpacked.

3. On the other hand, the operators + and / are the names of special variables in Common Lisp (Steele, p325), and hence cause no error when passed unpacked to the evaluator. Thus (LIST X + Y) returns a list of three elements, with no resort to DWIM; however, the parenthesized version (LIST (X + Y)) and the packed version (LIST X+Y) both work.

If you routinely DWIMIFY code, so that no CLISP infix forms (type 2 above) remain on your source files, you may not need to make any changes. However, note that the fact that DWIMIFY of prefix forms implicitly performed infix transformations can hide code that escaped being completely dwimified before being written to a file.

There is a further caution regarding even routinely dwimified code that has not been edited since before Koto. Two uses of the assignment operator (←) no longer work, if not explicitly dwimified, because their canonical form (the output of DWIMIFY) has changed, and the old form is no longer supported when the form is simply evaluated, macro-expanded, or compiled (with **DWIMIFYCOMPFLG** = **NIL**):

1. Iterative statement bindings must always be lists. For example, the old form

   ```
   (bind X←2 for Y in --)
   ```

   is now canonically

   ```
   (bind (X ← 2) for Y in --).
   ```

2. In a WITH expression, assignments must be dwimified to remove ←. For example, the old form

   ```
   (with MYRECORD MYFIELD ← (FOO))
   ```

   is now canonically

   ```
   (with MYRECORD (SETQ MYFIELD (FOO))).
   ```

DWIMIFY in Koto correctly made these transformations; however, in some older releases, it did not. Such old code must be explicitly dwimified (which you can do for these cases in Lyric). The errors resulting from failure to do so can be subtle. In particular, the compiler issues no special warning when such code is compiled. For example, in case 1, the macro expansion of the old form treats the symbol X←2 as a variable to bind, rather than as a binding of the variable X with initial value 2. The only hint from the compiler that anything is amiss is likely to be an indication that the variable X is used freely but not bound. Case 2 is even subtler: the symbols MYFIELD and ← are treated as symbols to be evaluated; since their values are not used, the compiler optimizes them away, reducing the entire expression to simply (FOO), and there is thus no warning of any sort from the compiler.

Chapter 22 Performance Issues

Section 22.3 Performance Measuring

(II:22.8)

The Interlisp-D **TIME** function has been withdrawn and replaced with the Common Lisp **TIME** macro (the symbol **TIME** is shared between IL and CL and thus need not be typed with a package prefix). The functionality of the *TIMEN* and *TIMETYP* arguments to the old **TIME** can be had by keywords to the **TIME** macro. The *Xerox Common Lisp Implementation Notes* describe the new **TIME** macro and its associated command in more detail.

Chapter 24 Streams and Files

The Xerox Common Lisp file system supports multiple streams open simultaneously on the same file. This is an *incompatible change* to the semantics of Interlisp-D. You may have to modify old programs if they have not followed the guidelines listed in Sec 24.5 of the *Interlisp-D Reference Manual*. Some of the implications of this change for Interlisp programs are described below.

In prior releases of Interlisp-D, the system treated the *name* of an open file as a synonym for the *stream* open on the file. This meant that only one stream could be open at any time on a given file. In the Lyric release, a file name is no longer a unique name for an open stream. Thus, file names are no longer acceptable as the file/stream argument to any input/output or file system function that operates on an open stream (**READ, PRINT, CLOSEF, COPYBYTES**, etc). The only non-stream values acceptable as stream designators are the symbols **NIL** and **T**, designating the primary and terminal input/output streams. An attempt to use a litatom, even a "full file name," as a stream designator signals the error "LITATOM 'streams' no longer supported." Strings no longer designate an input stream whose source is the string itself—programs should call **OPENSTRINGSTREAM** instead, or use the comparable Common Lisp forms, such as **CL:WITH-INPUT-FROM-STRING**.

The functions **OPENFILE** and **OPENSTREAM** are now synonymous—both return a stream instead of a "full file name." The functions **INPUT** and **OUTPUT** also return streams. One exception to this is that **INPUT** and **OUTPUT** return **T** in the case where the primary input or output stream was previously directed to the terminal. However, this special behavior is for the Lyric release only; we recommend that you convert old code that depended on testing **(EQ (OUTPUT) T)**. Note that the values of the variables ***STANDARD-INPUT*** and ***STANDARD-OUTPUT*** are always streams, even if they are directed to the terminal.

The function **FULLNAME** can be used to obtain the name of a stream. For your convenience, the print syntax of streams now includes the name of the stream (if to a file) and its access (input, output, etc.). Functions, such as **UNPACKFILENAME**, that manipulate file names generally accept a stream as well, extracting the name of the file from the stream.

INFILEP still returns a full file name, as it is merely recognizing a file, not opening a stream to it. Programmers should be wary of code that subsequently tries to use the value of **INFILEP** as a stream argument. And, of course, the *FILENAME* argument to **OPENSTREAM** is still a name (a symbol or string), not a stream. **OPENSTREAM** also accepts a Common Lisp pathname as its *FILENAME* argument.

The function **CLOSEALL** is no longer implemented. The function **OPENP** returns **NIL** when passed a file name (or anything else but

an open stream). However, for the Lyric release, **(OPENP NIL)** still returns a list of all streams open to files.

The functions **GETFILEINFO** and **SETFILEINFO** can still be given either an open stream or a file name. However, in the latter case, the request refers to the file, not to any stream open on the file. Thus, requesting the value of the attribute **LENGTH** for a file name may return a different value than requesting the value of the attribute **LENGTH** for a stream currently open on the file. **GETFILEINFO** returns **NIL** if given a file name and an attribute that only makes sense for streams (e.g., **ACCESS**, **ENDOFSTREAMOP**).

There is no difference between Common Lisp and Interlisp streams. A stream opened by an Interlisp function can be passed as argument to a Common Lisp input/output function, and vice versa.

Even though multiple streams per file are supported, the streams must still obey consistent access rules. That is, if a stream is open for output, no other streams on that file can be opened. It is not possible to **RENAMEFILE** or **DELFILE** a file that has *any* open stream on it.

The RS-232 or TTY ports are inherently single-user devices (rather than real files) thus, multiple streams cannot be open simultaneously on RS-232 or TTY.

Section 24.15 Deleting, Copying, and Renaming Files

(III:24.15)

The support of multiple streams per file now makes it possible to use **COPYFILE** without worrying about there being other readers of the file, in particular even when there is already a stream open on the file for sequential-only access (a case that failed in prior releases). Of course, **COPYFILE** cannot be used if the file already has an *output* stream open.

Chapter 25 Input/Output Functions

Variables Affecting Input/Output

There are several implicit parameters that affect the behavior of the input/output functions: the numeric print base, the primary output file, etc. In Common Lisp, these parameters are controlled by binding special variables. In Interlisp they are controlled by a functional interface; e.g., an output expression is wrapped in `(RESETFORM (RADIX 8) --)` to cause numbers to be printed in octal.

Where the input/output parameters in Common Lisp and Interlisp have essentially the same semantics, they have been integrated in Xerox Lisp. That is, binding the Common Lisp special variable and calling the Interlisp function are equivalent

operations, and they affect both Interlisp and Common Lisp input/output. However, it is considerably more efficient to bind a special variable than to call a function in a **RESETFORM** expression. In addition, binding a variable has only a local effect, whereas calling a function to side-effect the input/output parameters can also affect other processes. Thus, you are encouraged to use special variable binding to change parameters formerly changed via functional interface.

All of these variables are accessible in both the Common Lisp and Interlisp packages, so no package qualifier is required when typing them.

These parameters are as follows:

PRINT-BASE* vs RADIX** Binding ***PRINT-BASE to an integer *n* from 2 to 36 tells the printing functions to print numbers in base n. This is equivalent to (**RADIX** *n*). Note: this variable should not be confused with ***PRINT-RADIX***, another Common Lisp variable that controls whether Common Lisp functions include radix specifiers when printing numbers.

STANDARD-INPUT* vs INPUT** Binding ***STANDARD-INPUT to an input stream specifies the stream from which to read when an input function's stream argument is **NIL** or omitted. Evaluating ***STANDARD-INPUT*** is the same as evaluating (**INPUT**), except that (**INPUT**) returns **T** if the primary input for the process is the same as the terminal input stream (this compatibility feature is for the Lyric release only).

STANDARD-OUTPUT* vs OUTPUT** Binding ***STANDARD-OUTPUT to an output stream specifies the stream to which to print when an output function's stream argument is NIL or omitted. Evaluating ***STANDARD-OUTPUT*** is the same as evaluating (**OUTPUT**) except that (**OUTPUT**) returns **T** if the primary output for the process is the same as the terminal output stream (this compatibility feature is for the Lyric release only).

PRINT-LEVEL* & *PRINT-LENGTH* vs PRINTLEVEL** Binding ***PRINT-LEVEL to a positive integer *a* and ***PRINT-LENGTH*** to a positive integer *d* is equivalent to calling (**PRINTLEVEL** *a d*). Binding either variable to **NIL** is equivalent to specifying a value of -1 for the corresponding argument to **PRINTLEVEL**, i.e., it specifies infinite depth or length. Note that in Interlisp, print level is "triangular"—the print length decreases as the depth increases. In Common Lisp, the two are independent. Thus, although print level for both Interlisp and Common Lisp is controlled by a common pair of variables, the Interlisp and Common Lisp print functions interpret them (specifically ***PRINT-LENGTH***) slightly differently. In addition, Interlisp observes print level only when printing to the terminal, whereas Common Lisp observes it on all output.

READTABLE* vs SETREADTABLE** Binding ***READTABLE to a readtable specifies the readtable to use in any input/output function with a readtable argument omitted or specified as **NIL**. Evaluating ***READTABLE*** is the same as evaluating (**GETREADTABLE**). There is no longer a "NIL" or "T" readtable in Interlisp. See the discussion of readtables for more details.

Although the binding style is to be preferred to the **RESETFORM** expression, one difference should be noted with respect to error checking. In a form such as

(RESETFORM (RADIX *n*)
some-printing-code)

the value of *n* is checked immediately for validity, and an error is signalled if *n* is not an integer between 2 and 36. However, in

(LET ((*PRINT-BASE* *n*))
some-printing-code)

there is no error checking at the time of the binding; rather, an error will not be signalled until the code attempts to print a number.

Similarly, the values of ***STANDARD-INPUT*** and ***STANDARD-OUTPUT*** must be actual streams, not the values that print functions coerce to streams, such as **NIL**, **T** or window objects.

Integration of Common Lisp and Interlisp Input/output Functions

Common Lisp and Interlisp have slightly different rules for reading and printing, regarding such things as escape characters, case sensitivity and number format. Each has two kinds of printing function, an escaped version (intended for reading back in) and an unescaped version. In order that Common Lisp and Interlisp programs can more freely intermix, Xerox Lisp isolates most of the reading/printing differences in the readtables used by both languages, rather than in the functions themselves. The exact rules have been chosen as a reasonable compromise between backward compatibility with Interlisp and integration with Common Lisp. This section outlines the details of this integration.

In what follows, the phrase "the readtable" generally refers to the readtable in force for the read or print operation being discussed. Specifically, this means an explicit readtable (other than NIL or T) passed as readtable argument to an Interlisp function, or else the current binding of ***READTABLE***. See the section on readtables for more details.

Section 25.2 Input Functions

The functions **IL:READ** and **CL:READ**, given the same readtable, interpret an input in exactly the same way. That is, the functions obey Common Lisp syntax rules when given a Common Lisp readtable, and Interlisp syntax when given an Interlisp readtable. Thus, the principal difference between the two is in the functionality provided by **CL:READ**'s extra arguments: end of file handling and the ability to specify that the read is recursive, which is mostly important when reading input containing circular structure references (the ## and #= macros). See *Common Lisp, the Language* for details of **CL:READ**'s optional arguments.

There is one further difference between **IL:READ** and **CL:READ**, in the handling of the terminating character. If the read

terminates on a white space character, **CL:READ** consumes the character, while **IL:READ** leaves the character in the buffer, to be read by the next input operation. Thus, **IL:READ** is equivalent to **CL:READ-PRESERVING-WHITESPACE**. This difference is so that Interlisp code that calls (**READC**) following a (**READ**) of a symbol will behave consistently between Koto and Lyric.

The Interlisp function **SKREAD** now defaults its readtable argument to the current readtable, viz., the value of ***READTABLE***, rather than **FILERDTBL**. This makes it consistent with all the other input functions, and is usually the correct thing, especially with the new reader environments used by the File Manager, but it is an incompatible change from Koto. **SKREAD** is also now implemented using Common Lisp's ***READ-SUPPRESS*** mechanism, which means that, unlike in Koto, it does something reasonable when it encounters read macros.

Section 25.3 Output Functions

The discussion here is limited to the four basic printing functions: the unescaped and escaped Interlisp printing functions (**IL:PRIN1**, **IL:PRIN2**) and the corresponding Common Lisp functions (**CL:PRINC**, **CL:PRIN1**). All other print functions ultimately reduce to these. For example, **IL:PRINT** calls **IL:PRIN2**; **CL:FORMAT** with the ~S directive performs a **CL:PRIN1**.

IL:PRIN1 is essentially unchanged from previous releases. It uses no readtable at all, so is unaffected by the value of ***READTABLE***. It can be thought of as implicitly using the INTERLISP readtable.

Roughly speaking, **IL:PRIN2** and **CL:PRIN1** behave the same when given the same readtable. In particular, they both produce output acceptable to either **READ** function given the same readtable. Their minor differences are listed below.

CL:PRINC behaves like **CL:PRIN1**, except that it never prints escape characters or package prefixes. Thus, unlike **IL:PRIN1**, it *is* affected by the value of ***READTABLE***.

For the benefit of user-defined print functions, **IL:PRIN2** and **CL:PRIN1** bind ***PRINT-ESCAPE*** to **T**, while **IL:PRIN1** and **CL:PRINC** bind it to **NIL**. Thus, the print function can always examine ***PRINT-ESCAPE*** to decide whether it needs to print objects in a way that will read back correctly (Common Lisp user print functions may want to use **CL:WRITE** to pass ***PRINT-ESCAPE*** through transparently; Interlisp functions should choose **IL:PRIN2** or **IL:PRIN1** appropriately).

Printing Differences Between IL:PRIN2 and CL:PRIN1

There are two respects in which the Interlisp print functions (both **IL:PRIN1** and **IL:PRIN2**) differ from the Common Lisp ones, independent of readtable:

Line Length. The Interlisp functions respect the output stream's line length, while the Common Lisp functions all ignore it (they

never insert newline characters when output approaches the right margin).

Print Level. The Interlisp functions respect the print level variables only when printing to the terminal (unless **PLVLFILEFLG** is true, see the *Interlisp-D Reference Manual*) or when printing with a Common Lisp readtable, whereas the Common Lisp functions respect them on *all* output.

Internal Printing Functions

Interlisp has several functions (e.g., **NCHARS, STRINGWIDTH, CHCON, MKSTRING**) that operate on the "prin1 pname" of an object, or optionally on its "prin2 pname" when given an extra flag and readtable as arguments. These functions are essentially unchanged in Lyric.

In terms of the discussion above, the "prin1 pname" of an object continues to be the characters that would be produced by a call to **IL:PRIN1** at infinite print level and line length, and with ***PRINT-BASE*** bound to 10 (unless **PRXFLG** is true, see *Interlisp-d Reference Manual*). The "prin2 pname" of an object is the list of characters that would be produced by a call to **IL:PRIN2** (or **CL:PRIN1**) using the specified readtable (or ***READTABLE*** if none is given), again at infinite print level and line length.

Exception: the function **STRINGWIDTH** computes the width of the expression as if it were printed at the current ***PRINT-LEVEL*** and ***PRINT-LENGTH***.

Printing Differences between Koto and Lyric

The Common Lisp and Interlisp printing functions use the same strategy for escaping characters in symbol names. Because of this, symbols may print differently in Lyric than they did in Koto, for two reasons: the use of the Common Lisp multiple escape character, and the escaping of numeric print names. Although the appearance is different, the functionality is the same—symbols are still printed in a way that allows them to be correctly read.

Roughly speaking, the multiple escape character is used to escape symbol names that would require two or more single escape characters. Thus, for example, a symbol that printed as `%(OH% NO%)` in Koto will print in Lyric as `|(OH NO)|`. However, in the old readtables that lack a multiple escape character (e.g., OLD-INTERLISP-T), the single escapes are still used. Multiple escapes are also used to print a symbol containing lower-case letters when the readtable is case-insensitive, e.g., `|Small|` in a Common Lisp readtable. Keep in mind also that some additional characters are now "special", e.g., colon in all new readtables, semi-colon in Common Lisp. Thus, the typical NS File "full name" will be printed with the multiple escape character.

Since it is now possible to create symbols that have "numeric" print names, such symbols must be printed with suitable escape characters, so that on input they are not misinterpreted as numbers. For example, the symbol whose print name is "1.2E3"

is printed as |1.2E3|. In read tables lacking a multiple escape character, the single escape character is used instead, e.g., %1.2E3 in the old Interlisp T readtable. A print name is considered numeric according to the definition of "potential number" in Common Lisp (p. 341). Even if such a symbol is not readable in the current system as a number, it still needs to be escaped in case it is read into another system that treats it as numeric (either another Common Lisp implementation, or a future implementation of Xerox Lisp). Thus, some old Interlisp symbols now print escaped where they didn't in Koto; e.g., the **PRINTOUT** directive |.P2| is a potential number.

Bitmap Syntax

Bitmaps are printed in a new syntax in Lyric. When ***PRINT-ARRAY*** is **NIL** (the default at top level), a bitmap prints in roughly the same compact form as previously:

#<BITMAP @ *nn,nnnnnn*>

If ***PRINT-ARRAY*** is T, a bitmap prints in a manner that allows it to be read back:

#*(*Width Height* [*BitsPerPixel*])*XXXXXXXXX*...

Width and *Height* are measured in pixels; *BitsPerPixel* is supplied for bitmaps of other than the default of 1 bit per pixel. Each *X* represents four bits of a row of the bitmap; the characters @ and A through O are used in this encoding. Thus, there are 4*⌈*Width***BitsPerPixel*/16⌉ *X*'s for each row.

MAKEFILE binds ***PRINT-ARRAY*** to **T** so that bitmaps print readably in files. E.g., if the value of FOO is a bitmap, the command (VARS FOO) dumps something like

(RPAQQ FOO #*(10 10)ADSDKJFDKJH...)

Note that with this new format, bitmaps are readable even inside a complex list structure. This means you need no longer use the **UGLYVARS** command when dumping a list containing bitmaps if the bitmaps were previously the only "unprintable" part of the list.

Section 25.8 Readtables

(III:25.34)

The input/output syntaxes of Common Lisp and Interlisp differ in a few significant ways. For example, Common Lisp uses "\" as the escape character, whereas Interlisp uses "%". Common Lisp input is case-insensitive (lower-case letters are read as upper-case), whereas Interlisp is case-sensitive. In Xerox Lisp, these differences are accommodated by having different readtables for the two dialects. Which syntax is used for input or output depends on which readtable is being used (either as an explicit argument to the read/print function or by being the "current" readtable).

Interlisp readtables have been extended to include features of Common Lisp syntax. There is a registry of named readtables to make it easier to choose a readtable. The default Interlisp

readtable has been modified to make it look a little closer to Common Lisp.

Also, Xerox Lisp has a new mechanism for maintaining read/print consistency. This means that even though Koto files may contain characters that are now "special", such as colon, you need make no changes to them—the File Manager knows how to load them correctly. See Chapter 17, Reader Environments and File Manager for details of this mechanism.

Differences Between Interlisp and Common Lisp Read Tables

When reading or printing, the readtable dictates the syntax rules being followed. As in past releases, the readtable indicates which characters must be escaped when printing a symbol (and ***PRINT-ESCAPE*** is true). In addition, in Lyric the readtable specifies such things as which escape character to use (\ or %) and the package delimiter to print on package-qualified symbols. The less obvious rules are detailed below.

Printing numbers. Numbers are always printed in the current print base (the value of the variable ***PRINT-BASE***, or equivalently the value of **(RADIX)**). Whether to print a radix specifier is determined by the readtable. In Common Lisp, a radix specifier is printed exactly when the value of ***PRINT-RADIX*** is true. The radix specifier is a suffix decimal point in base 10, or a prefix using # for other bases. In Interlisp, a radix specifier is printed if the base is not 10, ***PRINT-ESCAPE*** is true, and the number is not less than the print base. The radix specifier is a suffix Q for octal, or a prefix using # (or | in old Interlisp readtables) for other bases. There is no decimal radix specifier.

Reading numbers. In Common Lisp, numbers are read in the current value of ***READ-BASE***, and a trailing decimal point is interpreted as a decimal radix specifier. In Interlisp, numbers are always read in base 10, and trailing decimal point denotes a floating-point number.

Case conversion. In a case-insensitive readtable (as Common Lisp is), the value of ***PRINT-CASE*** controls how upper-case symbols are printed, and lower-case letters in symbols are escaped. In a case-sensitive readtable (as Interlisp is), ***PRINT-CASE*** is ignored, so all letters in symbols are printed verbatim. ***PRINT-CASE*** is also ignored by **PRIN1**, which implicitly uses an Interlisp readtable.

Ratios. The character slash (/) is interpreted as the ratio marker in all readtables except old Interlisp readtables (specifically, those whose **COMMONNUMSYNTAX** property is **NIL**). This is so that old files containing symbols with slashes are not misinterpreted as ratios. Thus, the characters "1/2" are read in new readtables as the ratio 1/2, but in old Interlisp readtables as the 3-character symbol |1/2| (| is the multiple-escape character, see below). Ratios are printed in old Interlisp readtables in the form |.(/ numerator denominator).

Packages. Symbols are interned with respect to the current package (the binding of ***PACKAGE***) except in old Interlisp readtables (specifically, those whose **USESILPACKAGE** property is

T), where symbols are read with respect to the INTERLISP package, independent of the binding of ***PACKAGE***. Again, this is a backward-compatibility feature: Interlisp had no package system, so programmers were not confronted with the need to read and print in a consistent package environment.

Print Level elision. When ***PRINT-LEVEL*** or ***PRINT-LENGTH*** is exceeded, the printing functions denote elided elements and elided tails by printing "&" and "--" with an Interlisp readtable, or "#" and "..." with a Common Lisp readtable.

Section 25.8.2 New Readtable Syntax Classes

The following new syntax classes are recognized by **GETSYNTAX** and **SETSYNTAX**:

MULTIPLE-ESCAPE This character inhibits any special interpretation of all characters (except the single escape character) up until the next occurrence of the multiple escape character. In Common Lisp and in the new Interlisp readtables this character is the vertical bar ("|"). For example, |(a)| is read as the 3-character symbol "(a)"; |x\|y\\z| is read as the 5 character symbol "x|y\z".

There is no multiple escape character in the old Interlisp readtables.

PACKAGEDELIM This character separates a package name from the symbol name in a package-qualified symbol. In Common Lisp and in the new Interlisp readtables this character is colon (":"). In the old Interlisp readtables the package delimiter is control-↑ ("↑↑"); it is not intended to be easily typed, but exists only to have a compatible way to print package-qualified symbols in obsolete readtables. See *Common Lisp, the Language* for details of package specification.

Additional Readtable Properties

Read tables have several additional properties in Xerox Lisp. These are accessible via the function **READTABLEPROP**:

(READTABLEPROP *RDTBL PROP NEWVALUE*) [Function]

Returns the current value of the property *PROP* of the readtable *RDTBL*. In addition, if *NEWVALUE* is specified, the property's value is set to *NEWVALUE*. The following properties are recognized:

NAME The name of the readtable (a string, case is ignored). The name is used for identification when printing the readtable object itself, and can be given to the function **FIND-READTABLE** to retrieve a particular readtable.

CASEINSENSITIVE If true, then unescaped lower-case letters in symbols are read as upper-case when this readtable is in effect. This property is true by default in Common Lisp readtables and false in Interlisp readtables.

COMMONLISP If true, then input/output obeys certain Common Lisp rules; otherwise it obeys Interlisp rules. This is described in more detail in the section on reading and printing. Setting this property to

true also sets **COMMONNUMSYNTAX** true and **USESILPACKAGE** false.

COMMONNUMSYNTAX If true, then the Common Lisp rules for number parsing are followed; otherwise the old Interlisp rules are used. This affects the interpretation of "/" and the floating-point exponent specifiers "d", "f", "l" and "s". It does not affect the interpretation of decimal point and ***READ-BASE***, which are controlled by the **COMMONLISP** property. **COMMONNUMSYNTAX** is true for Common Lisp readtables and the new Interlisp readtables; it is false for old Interlisp readtables.

USESILPACKAGE This is a backward compatibility feature. If **USESILPACKAGE** is true, then the Interlisp input/output functions read and print symbols with respect to the Interlisp package, independent of the current value of ***PACKAGE***. This property is true by default for old Interlisp readtables and false for others.

The following properties let the print functions know what characters are being used for certain variable syntax classes so that they can print objects in a way that will read back correctly. Note that it is possible for several characters to have the same syntax on input, but only one of the characters is used for output. Also note that only the three syntax classes **ESCAPE**, **MULTIPLE-ESCAPE** and **PACKAGEDELIM** are parameterized for output; the others (such as **LEFTPAREN** and **STRINGDELIM**) are hardwired —the same character is always used.

ESCAPECHAR This is the character code for the character to use for single escape. Setting this property also gives the designated character the syntax **ESCAPE** in the readtable.

MULTIPLE-ESCAPECHAR This is the character code for the character to use for multiple escape. Setting this property also gives the designated character the syntax **MULTIPLE-ESCAPE** in the readtable.

PACKAGECHAR This is the character code for the package delimiter. Setting this property also gives the designated character the syntax **PACKAGEDELIM** in the readtable.

(**FIND-READTABLE** *NAME*) [Function]

Returns the readtable whose name is *NAME*, which should be a symbol or string (case is ignored); returns **NIL** if no such readtable is registered. Readtables are registered by calling (**READTABLEPROP** *rdtbl* 'NAME *name*).

(**COPYREADTABLE** *RDTBL*) [Function]

COPYREADTABLE has been extended to accept a readtable name as its *RDTBL* argument (the old value ORIG could be considered a special case of this). For example, (**COPYREADTABLE** "INTERLISP") returns a copy of the INTERLISP readtable. **COPYREADTABLE** preserves all syntax settings and properties except NAME.

Section 25.8 Predefined Readtables

The following readtables are registered in the Lyric release of Xerox Lisp:

INTERLISP This is the "new" Interlisp readtable. It is used by default in the Interlisp Exec and by the File Manager to write new versions of pre-existing source files. It thus replaces the old T readtable, FILERDTBL, CODERDTBL and DEDITRDTBL. It differs from them in the following ways:

|(vertical bar) has syntax **MULTIPLE-ESCAPE** rather than being used as a variant of the Common Lisp dispatching # macro character.

is used as the Common Lisp dispatching # macro character. For example, to type a number in hexadecimal, the syntax is #xnnn rather than |xnnn.

:(colon) has syntax **PACKAGEDELIM**.

'(quote) reads the next expression as (QUOTE expression).

`(backquote)
,(comma) are used to read backquoted expressions

In addition, the Common Lisp syntax for numbers is supported (the readtable has property **COMMUNNUMSYNTAX**). For example, the characters "1/2" denote a ratio, not a symbol. Note, however, that trailing decimal point still means floating point, rather than forcing a decimal read base for an integer.

The syntax for quote, backquote, and comma is the same as in OLD-INTERLISP-T, so you will not see any difference when typing to an Interlisp Exec. However, the fact that files are also written in the new INTERLISP readtable means that the prettyprinter is now able to print quoted and backquoted expressions much more attractively on files (and to the display as well).

LISP This readtable implements Common Lisp read syntax, exactly as described in *Common Lisp, the Language*.

XCL This readtable is the same as LISP, except that the characters with ASCII codes 1 thru 26 have white-space (**SEPRCHAR**) syntax. This readtable is intended for use in File Manager files, so that font information can be encoded on the file.

The following readtables are provided for backward compatibility. They are the same as the corresponding readtables in the Koto release, with the addition of the **USESILPACKAGE** property.

ORIG This is the same as the ORIG readtable described in the *Interlisp-D Reference Manual*. When using a readtable produced by (**COPYREADTABLE** 'ORIG), expressions will read and print exactly the same in Koto and Lyric.

OLD-INTERLISP-FILE This is the same as the FILERDTBL described in the *Interlisp-D Reference Manual*. This readtable is used to read source files produced in the Koto release. Note that in Lyric, FILERDTBL is no longer used when reading or writing new files; see the section on reader environments.

OLD-INTERLISP-T This is the same as the T readtable described in the *Interlisp-D Reference Manual*.

If you wish to change the syntax used by a standard readtable, it is recommended instead that you copy the readtable, give it a

distinguished name, and make the change in the new readtable. This will reduce the likelihood that you will try to read another user's files in an incompatible readtable, or that another user will fail reading yours. See chpater 17, Reader Environments and the File Manager, for more details.

Koto Compatibility Considerations

In order to consistently read a data structure that you have previously printed, it is important that **READ** and **PRINT** both use the same readtable and package. Code that calls **READ** or **PRINT** without explicitly specifying a readtable (via the *RDTBL* argument or by doing a **SETREADTABLE**) is thus in some danger of reading and printing inconsistently.

Specifying Readtables and Packages

In Koto, the "primary" (NIL) readtable was not significantly different from the other Interlisp readtables, and users tended not to make significant modifications to the primary readtable anyway. As a result, it was easy to write code that was not careful about readtables and get away with it. In Lyric, however, there are significant differences among commonly used readtables. Thus, if code using the default readtable called **PRINT** under, say, the Common Lisp Executive and tried to **READ** the expression back while running under an Interlisp Executive, it might very well get inconsistent results.

Lyric also introduces the extra complication of the default package, which is the other important parameter affecting the behavior of **READ** and **PRINT**.

Programmers are thus advised to fix any code that uses **READ** and **PRINT** as a way of storing and retrieving Lisp expressions to be sure to specify a readtable and package environment. For new code in Lyric, this can be done by binding the special variables ***READTABLE*** and ***PACKAGE***. If it is necessary to write code that works in both Koto and Lyric, the programmer should pass an explicit readtable to all **READ** and **PRINT** functions, or set the primary readtable using (**RESETFORM** (**SETREADTABLE** *rdtbl*) --). If the readtable chosen is either FILERDTBL or one derived as a copy of ORIG, then **READ** and **PRINT** will automatically use the INTERLISP package in Lyric, thereby avoiding any need to specify a binding for ***PACKAGE***.

The T Readtable

An additional possible incompatibility exists with regard to the Koto T readtable: The T readtable was "the readtable used when reading from the terminal". In Lyric, the T readtable is synonymous with NIL, and all Executives bind ***READTABLE*** to the appropriate value for the Exec. This is unlikely to be a major source of incompatibility, as few programs depend on printing something in the T readtable in a way that needs to read back consistently.

PQUOTE Printed Files

In Lyric, the prettyprinter automatically prints quoted and backquoted expressions attractively. Hence, the PQUOTE Lispusers module is now obsolete. However, if you have written files in the past with the PQUOTE module loaded into your environment, you need to do the following in Lyric in order to load those files:

(SETSYNTAX (CHARCODE "'") '(MACRO FIRST READQUOTE) FILERDTBL)

You can then load the old files. New files produced in Lyric by **MAKEFILE** will automatically be loadable, so you need only perform the **SETSYNTAX** change as long as you still have old files written with PQUOTE. Remember, of course, that as long as the **SETSYNTAX** is in effect (as with the old PQUOTE module), if you read old files that were written without PQUOTE you may read them incorrectly.

Back-Quote Facility

The back-quote facility now fully conforms with *Common Lisp the Language*. This means some cases of nested back-quote now work correctly. Back-quote forms are also more attractively displayed by the prettyprinter. Users should beware, however, that the back-quote facility does not attempt to create fresh list structures unless it is necessary to do so. Thus for example,

```
'(1 2 3)
```

is equivalent to

```
'(1 2 3)
```

not

```
(LIST 1 2 3)
```

If you need to avoid sharing structure you should explicitly use **LIST**, or **COPY** the output of the back-quote form.

[This page intentionally left blank]

4. CHANGES TO INTERLISP-D SINCE KOTO

This section contains release notes indicating changes that have occurred in Interlisp-D since the Koto release. These changes are generally unrelated to the integration of Common Lisp in the Xerox Lisp environment. Organization of this section corresponds to the *Interlisp-D Reference Manual*.

VOLUME I—LANGUAGE

Chapter 3 Lists

Section 3.2 Building Lists From Left To Right

(I:3.7)

The functions **DOCOLLECT** and **ENDCOLLECT** are no longer supported.

(I:3.8)

The description of the **ADDTOSCRATCHLIST** function has been revised to read:

(ADDTOSCRATCHLIST *VALUE*) [Function]

For use inside a **SCRATCHLIST** form. *VALUE* is added on to the end of the value being collected by **SCRATCHLIST**. When the **SCRATCHLIST** returns, its value is a list containing all of the things that **ADDTOSCRATCHLIST** has added.

Section 3.10 Sorting Lists

(I:3.17)

(SORT *DATE COMPAREFN*) [Function]

There is no safe interrupt to **SORT**—if you abort a call to **SORT** by *any* means the possibility exists for losing elements from the list being sorted.

Chapter 6 Hash Arrays

(I:6.1)

(HASHARRAY *MINKEYS OVERFLOW HASHBITSFN EQUIVFN RECLAIMABLE REHASH-THRESHOLD*) [Function]

The function **HASHARRAY** has two new optional arguments, *RECLAIMABLE* and *REHASH-THRESHOLD*. If *RECLAIMABLE* is true, then entries in the hash table are considered "reclaimable" in the sense that the system is permitted to remove any key and

its associated value from the hash table at any time. In practice, the contract is less severe: the system only removes keys when a hash table fills and is about to be rehashed, and then it only removes keys whose reference count is one, and to which there are thus no pointers outstanding except possibly from the stack (local variables). This is useful for hash tables that serve to cache information about Lisp objects to avoid recomputation; for example, the system hash table CLISPARRAY is now reclaimable. Discarding keys keeps the table from necessarily needing to grow, and potentially allows the storage consumed by both the key and value to be reclaimed.

Section 6.1 Hash Overflow

(I:6.3)

You should note changes to the wording of two of the possibilites for the overflow method:

The first sentence for **NIL** should read: The array is automatically enlarged by *at least* a factor of 1.5 every time it overflows.

The explanation for "a positive integer N" should read: The array is enlarged to include *at least* N more slots than it currently has.

Chapter 7 Integer Arithmetic

(I:7.5)

The variables **MIN.INTEGER** and **MAX.INTEGER** have been removed from the *Interlisp-D Reference Manual*. Therefore, calling **(MIN)** and **(MAX)** is an error.

(I:7.7)

(**FIXR** *N*) [Function]

When N is exactly half way between two integers, FIXR rounds it to the even number. For example (FIXR 1.5) ⇒ 2 and (FIXR 2.5) ⇒ 2.

Section 7.3 Logical Arithmetic Functions

The function **INTEGERLENGTH** does *not* coerce floating point numbers to integers; rather, it signals an error, "Arg not Integer". (This was true in Koto as well.)

Section 7.5 Other Arithmetic Functions

(I:7.13)

The algorithms for **SIN**, **COS**, and other trigometric functions have been tuned and are now accurate to at least six significant figures.

Chapter 9 Conditionals and Iterative Statements

Section 9.2 Equality Predicates

(I:9.3)

(EQUALALL *X Y*) [Function]

Add the following NOTE to the **EQUALALL** function:

Note: In general, **EQUALALL** descends all the way into all datatypes, both those defined by the user and those built into the system. If you have a data structure with fonts and pointers to windows, **EQUALALL** will descend into those also. If the data structures are circular, as windows are, **EQUALALL** can cause a Stack Overflow error.

Section 9.8.3 Condition I.s. oprs

UNTIL *N* (*N* a number) [I.S. Operator]

REPEATUNTIL *N* (*N* a number) [I.S. Operator]

These descriptions were included in the Interlisp-D Reference Manual in error and should be removed. **UNTIL** and **REPEATUNTIL** work *only* with predicate expressions, not numbers.

Chapter 10 Function Definition, Manipulation , and Evaluation

Section 10.2 Defining Functions

(I:10.11)

In the definition of the **MOVD** function, the sentence "**COPYDEF** is a higher-level function that only moves expr definitions, but..." should be revised to read:

COPYDEF is a higher-level function that not only moves expr definitions, but also works for variables, records, etc.

Section 10.5 Functional Arguments

(I:10.19)

FUNARG functionality (non-**NIL** second argument to **FUNCTION**) has been withdrawn. Most of the uses for Interlisp **FUNARG**'s are better written using the lexical closure functionality of Common Lisp.

Section 10.6.2 Interpreting Macros

The variables **SHOULDCOMPILEMACROATOMS** and **UNSAFEMACROATOMS** no longer exist.

Chapter 11 Variable Bindings and the Interlisp Stack

(II:11.2)

In Xerox Lisp there is a fixed amount of space allocated for the stack. When this space is exhausted, the **STACK OVERFLOW** error occurs. However, if the system waited until the stack were *really* exhausted, there wouldn't be room to run the debugger. Thus, a portion of the stack space is reserved; when the stack intrudes into the reserved area, it causes a stack overflow interrupt, and subsequently a call to the debugger.

In order not to get a **STACK OVERFLOW** error while inside the debugger, this intrusion into the reserved area is only noted once. If the reserved area is exhausted, then a "hard" stack overflow occurs (a 9319 MP halt), from which the only recourse is a hard reset via STOP (or Ctrl-D from TeleRaid). Following a hard reset, the stack is cleared, stack overflow detection is reenabled, and all processes are restarted.

The implications of this are that you should not attempt any deep computations while inside the debugger for a stack overflow error, and you should call **(HARDRESET)** as soon as possible in order that subsequent stack overflows can again be caught in the debugger before they advance to the MP halt. In order to make this more convenient, the system automatically calls **(HARDRESET)** if you exit the debugger via the ↑ or **OK** commands, or abort with a Ctrl-D. The only way to exit the debugger without having a **(HARDRESET)** occur is by using the **RETURN** command. You can disable this feature by setting **AUTOHARDRESETFLG** to **NIL**, in which case you must be sure to perform the **(HARDRESET)** yourself if you want the next stack overflow to be detected early enough to enter the debugger.

Section 11.2.1 Searching the Stack

(**STKPOS** *FRAMENAME N POS OLDPOS*) [Function]

(STKPOS 'STKPOS) does not cause an error; it merely returns NIL. (This was true in Koto as well.) It is still not permissible to create a pointer to the active frame; however, **STKPOS** never attempts to, as it starts searching for the specified frame by looking first at its caller.

Section 11.2.2 Variable Bindings in Stack Frames

(I:11.7)

(**STKARG** *N POS* —) [Function]

(**STKNARGS** *POS* —) [Function]

The functions **STKARG** and **STKNARGS** will now return the number of arguments supplied to a Lambda Nospread when there is a break. The **?=** command will show all the arguments.

(**SETSTKARGNAME** *N POS NAME*) [Function]

The function **SETSTKARGNAME** does not work for interpreted frames.

Section 11.2.5 Releasing and Reusing Stack Pointers

(CLEARSTK *FLG*) [Function]

(CLEARSTK NIL) is a no-op—the ability to clear all stack pointers is inconsistent with the modularity implicit in a multi-processing environment.

CLEARSTKLST [Variable]

NOCLEARSTKLST [Variable]

The variables CLEARSTKLST and NOCLEARSTKLST are no longer used. (More precisely, they are used only by the Old Interlisp Executive, which means that programs can no longer depend on them.)

Section 11.2.7 Other Stack Functions

(II:11.13)

In the **REALFRAMEP** function, the *INTERPFLG* argument description has been corrected to read:

If *INTERPFLG* = **T** returns **T** if *POS* is not a dummy frame. For example, if (**STKNAME** *POS*) = **COND**, (**REALFRAMEP** *POS*) is **NIL**, but (**REALFRAMEP** *POS* **T**) is **T**.

Chapter 12 Miscellaneous

Section 12.2 Idle Mode

The following properties in **IDLE.PROFILE** are new or have meanings different from the documentation in the *Interlisp-D Reference Manual*:

ALLOWED.LOGINS The authentication aspects of this property have been separated into the **AUTHENTICATE** property. The value of this property now speaks specifically to who is allowed to exit idle mode: If the value is **NIL** (or any other non-list), no login at all is required to exit Idle mode. Otherwise, the value is a list composed of any of the following:

***** Require login, but let anyone exit idle mode. This will overwrite the previous user's name and password each time idle mode is exited.

T Let the previous user (as determined by **USERNAME**) exit idle mode. If the user name has not been set, this is equivalent to *****.

A user name Let this specific user exit idle mode.

A group name Allow any members of this group (an NS Clearinghouse group name) to exit idle mode.

The initial value for **ALLOWED.LOGINS** is (**T ***), i.e., anyone is allowed to exit idle mode.

AUTHENTICATE The value of this property determines what mechanism is used to check passwords. If **T**, use the NS authentication protocol (requires the presence of an NS Authentication server accessible via the network). If **NIL**, do not check the password at all—accept any password. This is only particularly useful if **ALLOWED.LOGINS** contains ***.**

The initial value of **AUTHENTICATE** is **T**.

FORGET If this is the symbol **FIRST**, the user's password will be erased when idle mode is entered. Otherwise, this property is relevant only when **ALLOWED.LOGINS** is **NIL** (if **ALLOWED.LOGINS** is a list, then some sort of login is required, which will have the effect of erasing any previous login): If **FORGET** is non-**NIL**, the user's password will be erased when idle mode is exited. Initial value is **T** (erase password on exit).

Note: If the password is erased on *entry* to Idle mode (value **FIRST**), any program left running when idle mode is entered may fail if it tries doing anything requiring passwords (such as accessing file servers).

LOGIN.TIMEOUT Value is a number indicating how many seconds Idle's prompt for a login should remain up before timing it out and resuming Idle mode. Initial value is 30. This feature avoids the problem of having an Idle machine "freeze up" indefinitely (stop running the idle pattern) just because someone brushed against the keyboard.

RESETVARS This property is no longer used; rather, the value of the global variable **IDLE.RESETVARS** is used instead.

SUSPEND.PROCESS.NAMES This property is no longer used; rather, the value of the global variable **IDLE.SUSPEND.PROCESS.NAMES** is used instead.

Section 12.3 Saving Virtual Memory State

AROUNDEXITFNS [Variable]

This variable provides a way to "advise" the system on cleanup and restoration activities to perform around **LOGOUT**, **SYSOUT**, **MAKESYS** and **SAVEVM**; it subsumes the functionality of **BEFORESYSOUTFORMS**, **AFTERLOGOUTFORMS**, etc. Its value is a list of functions (names) to call around every "exit" of the system. Each function is called with one argument, a symbol indicating which particular event is occurring:

BEFORELOGOUT The system is about to perform a **LOGOUT**. The event function might want to save state, notify a network connection that it is about to go away, etc.

BEFORESYSOUT
BEFOREMAKESYS
BEFORESAVEVM The system is about to perform a **SYSOUT**, **MAKESYS**, or **SAVEVM**. Often these three events are treated equivalently; however, sometimes the distinction is interesting. For example, a program might want to perform a much more extensive tidying-up before **MAKESYS** than if it is merely doing a routine **SAVEVM**.

AFTERLOGOUT
AFTERSYSOUT
AFTERMAKESYS
AFTERSAVEVM

The system is starting up a virtual memory image that was saved by performing a **LOGOUT**, **SYSOUT**, etc. Ordinarily, the event function should treat all of these the same—in all four cases, some arbitrary amount of time has passed, remote files may have come and gone, a different user may be logged in, or the virtual memory image might even be running on a different workstation.

AFTERDOSYSOUT
AFTERDOMAKESYS
AFTERDOSAVEVM

The system is continuing in the same virtual memory image following a **SYSOUT**, **MAKESYS**, or **SAVEVM** (as opposed to having just booted the same virtual memory image). Ordinarily, these events are uninteresting; they exist solely so that actions taken by the **BEFORExxx** events can be compensated for after the event. For example, if the before event cleared a cache, the after event might initiate refilling it. There is, of course, no event **AFTERDOLOGOUT**, as **LOGOUT** does not "continue".

Section 12.4 System Version Information

(I:12.13)

In the description of the **MACHINETYPE** function, add another machine, the **DOVE** (for the Xerox 1186).

VOLUME II—ENVIRONMENT

Chapter 13 Interlisp Executive

(I:23.37)

(**READLINE** *RDTBL* — —) [Function]

The *Interlisp-D Reference Manual* states:

The description on p 13.37 of **READLINE**'s behavior when one or more spaces precede the carriage return applies only when **LISPXREADFN** is **READ**. **LISPXREADFN** is initially **TTYINREAD**, which ignores spaces before the carriage return, and thus will never prompt you with "..." for an additional line. Also, the new Executive does not use **READLINE** at all, so you will never see this behavior in a new Executive, independent of the setting of **LISPXREADFN**.

Chapter 14 Errors and Breaks

Section 14.5 Break Window Variables

(II:14.15)

Setting the variable **BREAKREGIONSPEC** to NIL no longer creates problems if there is a subsequent break.

Section 14.8 Catching Errors

(II:14.22)

The Nlambda functions **ERSETQ** and **NLSETQ** now allow evaluation of an arbitrary number of forms, rather than only one.

Chapter 17 File Package

Note: The File Package is now known as the File Manager.

Section 17.8.1 Functions for Manipulating Typed Definitions

(II:17.26)

(**HASDEF** *NAME TYPE SOURCE SPELLFLG*) [Function]

Clarification: **HASDEF** for type FNS (or NIL) indicates that *NAME* has an editable source definition, not that *NAME* is defined at

all. Thus if *NAME* exists on a file for which you have loaded only the compiled version but not the source, **HASDEF** returns **NIL**.

Section 17.8.2 Defining New File Package Types

(II:17.31)

In the **WHENCHANGED** File Package Type Property the *REASON* argument passed to **WHENCHANGED** no longer is **T** or **NIL**. The Note has been revised as follows:

Note: The *REASON* argument passed to **WHENCHANGED** functions is either **CHANGED** or **DEFINED**.

(II:17.32)

The Nospread Function **FILEPKGTYPE** returns a property list rather than an alist.

Section 17.9.8 Defining New File Package Commands

(II:17.47)

The Nospread Function **FILEPKGCOM** returns a property list rather than an alist.

Section 17.11 Symbolic File Format

(**PRETTYDEF** *PRTTYFNS PRTTYFILE PRTTYCOMS REPRINTFNS SOURCEFILE CHANGES*) [Function]

PRETTYDEF accepts only a symbol for its file argument.

(**LISPSOURCEFILEP** *FILE*) [Function]

LISPSOURCEFILEP is more specifically defined to mean that the file is in File Manager format *and* has a file map.

Section 17.11.3 File Maps

File maps are no longer stored on the FILEMAP property. See **GET-ENVIRONMENT-AND-FILEMAP** in Chapter 3 Integration of Interlisp-D and Common Lisp, "Programmer's Interface to Reader Environments."

Chapter 18 Compiler

Note that you should not attempt to compile a file containing a function named **STOP**. The format of the .LCOM file produced by **BCOMPL** or **TCOMPL** admits an unfortunate ambiguity in the treatment of the symbol **STOP**—**LOAD** prefers to treat it as the token signifying the end of the file, rather than as starting the definition of the function **STOP**.

There is no such restriction for the .DFASL files produced by **CL:COMPILE-FILE**.

Chapter 21 CLISP

Section 21.8 Miscellaneous Functions and Variables

(CLEARCLISPARRAY *NAME* — —) [*Function*]

Macro and CLISP expansions are cached in CLISPARRAY, the system's CLISP hash array. When anything changes that would invalidate an expansion, it needs to be removed from the cache. CLEARCLISPARRAY removes from the CLISP hash array any key whose CAR is *NAME*. The system does this automatically whenever you edit a clisp or macro form, or when you redefine a clisp word or macro definition. However, you may need to call CLEARCLISPARRAY explicitly if you change something in a more subtle way, e.g., you redefine a function used by a macro. If your change invalidates an unknown set of expansions, you might prefer to take the performance penalty of calling (CLRHASH CLISPARRAY) to invalidate the entire cache, just to make sure no incorrect expansions are kept around.

Chapter 22 Performance Issues

Section 22.1 Storage Allocation and Garbage Collection

The following should be appended to the description of garbage collection in Interlisp-D:

Another limitation of the reference-counting garbage collector is that the table in which reference counts are maintained is of fixed size. For typical Lisp objects that are pointed to from exactly one place (e.g., the individual conses in a list), no burden is placed on this table, since objects whose reference count is 1 are not explicitly represented in the table. However, large, "rich" data structures, with many interconnections, backward links, cross references, etc, can contribute many entries to the reference count table. For example, if you created a data structure that functioned as a doubly-linked list, such a structure would contribute an entry (reference count 2) for each element.

When the reference count table fills up, the garbage collector can no longer maintain consistent reference counts, so it stops doing so altogether. At this point, a window appears on the screen with the following message, and the debugger is entered:

> Internal garbage collector tables have overflowed, due
> to too many pointers with reference count greater than 1.
> *** The garbage collector is now disabled. ***
> Save your work and reload as soon as possible.

[This message is slightly misleading, in that it should say "count not equal to 1". In the current implementation, the garbage collection of a large pointer array whose elements are not otherwise pointed to can place a special burden on the table, as each element's reference count simultaneously drops to zero and

is thus added to the reference count table for the short period before the element is itself reclaimed.]

If you exit the debugger window (e.g., with the RETURN command), your computation can proceed; however, the garbage collector is no longer operating. Thus, your virtual memory will become cluttered with objects no longer accessible, and if you continue for long enough in the same virtual memory image you will eventually fill up the virtual memory backing store and grind to a halt.

Section 22.5 Using Data Types Instead of Records

(II:22.13)

The note in this section states that "pages for datatypes are allocated one page at a time." The note should read:

Space for datatypes is allocated two pages at a time. Thus, each datatype for which any instances at all have been allocated has at least two pages assigned to it.

Chapter 23 Processes

Section 23.6 Typein and the TTY Process

BACKGROUNDFNS [Variable]

A list of functions to call "in the background". The system runs a process (called "BACKGROUND") whose sole task is to call each of the functions on the list **BACKGROUNDFNS** repeatedly. Each element is the name of a function of no arguments. This is a good place to put cheap background tasks that only do something once in a while and hence do not want to spend their own separate process on it. However, note that it is considered good citizenship for a background function with a time-consuming task to spawn a separate process to do it, so that the other background functions are not delayed.

TTYBACKGROUNDFNS [Variable]

This list is like **BACKGROUNDFNS**, but the functions are only called while in a tty input wait. That is, they always run in the tty process, and only when the user is not actively typing. For example, the flashing caret is implemented by a function on this list. Again, functions on this list should spend very little time (much less than a second), or else spawn a separate process.

VOLUME III—INPUT/OUTPUT

Chapter 24 Streams and Files

Section 24.7 File Attributes

(**GETFILEINFO** *FILE ATTRIB*) [Function]

NS file servers implement the following additional attributes for **GETFILEINFO** (neither of these attributes is currently settable with **SETFILEINFO**):

READER The name of the user who last read the file.

PROTECTION A list specifying the access rights to the file. Each element of the list is of the form (*name nametype . rights*), where *name* is the name of a user or group or a name pattern, and *rights* is one or more of the symbols ALL READ WRITE DELETE CREATE MODIFY. For servers running Services release 10.0 or later, *nametype* is the symbol "--"; in earlier releases it is either INDIVIDUAL or GROUP, to distinguish the type of name. For example, the value ((Jane Jones: -- ALL) (*: -- READ)) means that user Jane Jones has full access to the file, while all members of the default domain only have read access to the file.

Section 24.18.1 Pup File Server Protocols

UNIXFTPFLG [Variable]

When the Leaf protocol was first implemented for the Vax Unix operating system, its use was inconsistent with the operation of the Pup Ftp server on the same host: the Leaf server supported versions, but the Ftp server knew only about the native, versionless file system. Thus, Lisp could not use the two protocols interchangeably. For example, if it used Ftp to write a file FOO, the Ftp server would, in versionless style, overwrite the versionless file FOO, rather than create a new version FOO;6 to supersede the highest version FOO;5 created by the Leaf server.

Lisp thus makes the conservative assumption that the Ftp server is unusable for anything other than directory enumeration on a host of type UNIX. This is unfortunate, since it is often the case that Ftp is more efficiently implemented than Leaf, since one need only tune the performance of sequential access.

More recent versions of the Unix Pup software have a Leaf and Ftp server more in agreement with each other. Setting **UNIXFTPFLG** to true (it is initially NIL) informs Lisp that all the Unix servers accessible on your internetwork that possess Ftp servers are safe to use in parallel with their Leaf servers.

Section 24.18.3 Operating System Designations

DEFAULT.OSTYPE [Variable]

If a host's name is not found in **NETWORKOSTYPES**, its operating system type is assumed to be the value of **DEFAULT.OSTYPE**. This variable may be of use to sites with many servers all of the same type. Its default value (IFS) is, unfortunately, inappropriate for most sites. It is recommended you set **DEFAULT.OSTYPE** in the initialization file that lives on the local disk (*not* in an init file on a file server, since Lisp needs to know the operating system type before talking to the server).

Chapter 25 Input/Output Functions

Section 25.2 Input Functions

(LASTC *FILE***)** [Function]

The function **LASTC** can return an incorrect result when called immediately following a **PEEKC** on a file that contains run-coded NS characters.

Section 25.3.2 Printing Numbers

(III:25.15)

In the **PRINTNUM** function, the **FLOAT** format option **(FLOAT 7 2 NIL T)** is illegal; change the option to **(FLOAT 7 2 NIL 0)**.

Section 25.3.4 Printing Unusual Data Structures

(HPRINT *EXPR FILE UNCIRCULAR DATATYPESEEN***)** [Function]

Using **HPRINT** to save structures that include pointers to raw storage will cause stack overflows. This includes dumping things using the **VARS**, **UGLYVARS**, or **HORRIBLEVARS** filemanager commands.

For example, a font descriptor points to raw storage, and cannot be dumped; for that reason, other system data types (e.g. windows) that point to fonts also cannot be dumped.

Section 25.4 Random Access File Operations

(III:25.20)

The first argument in the **FILEPOS** function should be called *STR* not *PATTERN*.

Section 25.6 PRINTOUT

(III:25.27)

The PRINTOUT command **.FONT** changes the **DSPFONT** font permanently, that is, even after printout finishes.

Section 25.8.3 READ Macros

(III:25.42-43)

These **READMACROS** appear only in the OLD-INTERLISP-T readtable. (See Section 2 for a description of Lyric readtables.)

Chapter 26 User Input/Output Packages

Section 26.3 ASKUSER

(**ASKUSER** *WAIT DEFAULT MESS KEYLST TYPEAHEAD LISPXPRNTFLG OPTIONSLST FILE*) [Function]

ASKUSER does not accept a string to mean a stream open on the string; you must call **OPENSTRINGSTREAM** if that's what you mean.

Section 26.4.5 Useful Macros

(III:26.29)

CTRLUFLG is no longer supported by default. To use this feature, turn it on explicitly: **(INTERRUPTCHAR (CHARCODE ↑U) 'CTRLUFLG)**.

Chapter 27 Graphic Output Operations

Section 27.1.3 Bitmaps

Note: The printed representation of bitmpas has changed. Please see release notes Chapter 3, Integration of Interlisp-D/ Common Lisp, "Bitmap Syntax"

(III:27.4)

The following function has been added to Bitmap Operations between the functions **EXPANDBITMAP** and **SHRINKBITMAP**:

(**ROTATE-BITMAP** *BITMAP*) [Function]

Given an m-high by n-wide bitmap, this function returns an n-high by m-wide bitmap. The returned bitmap is the image of the original bitmap, rotated 90 degrees clockwise.

Section 27.3 Accessing Image Stream Fields

The following functions were not documented in the Koto release of the Interlisp-D reference Manual:

(**DSPCLEOL** *XPOS YPOS HEIGHT*) [Function]

"Clear to end of line". Clears a region from (*XPOS,YPOS*) to the right margin of the display, with a height of *HEIGHT*. If *XPOS* and *YPOS* are **NIL**, clears the remainder of the current display line, using the height of the current font.

(**DSPRUBOUTCHAR** *DS CHAR X Y TTBL*) [Function]

Backs up over character code *CHAR* in the display stream *DS*, erasing it. If *X*, *Y* are supplied, the rubbing out starts from the position specified. **DSPRUBOUTCHAR** assumes *CHAR* was printed with the terminal table *TTBL*, so it knows to handle control characters, etc. *TTBL* defaults to the primary terminal table.

Section 27.6 Drawing Lines

(III:27.18)

The **RELDRAWTO** function has been corrected so that it no longer draws a spot if the *DX* and *DY* arguments are 0.

Section 27.7 Drawing Curves

(III:27.18)

For the brush width value of **NIL**, the previous default value **(ROUND 1)** has been changed. The default value for the brush width value **NIL** is the **DSPSCALE** of the stream (that is, 1 printer's point wide).

(III:27.19)

A new image stream function, **DRAWARC**, follows **DRAWCIRCLE** in the *InterLisp-D Reference Manual*.

(**DRAWARC** *CENTERX CENTERY RADIUS STARTANGLE NDEGREES BRUSH DASHINGSTREAM*) [Function]

Draws an arc of the circle whose center point is (*CENTERX CENTERY*) and whose radius is *RADIUS* from the position at *STARTANGLE* degrees for *NDEGREES* number of degrees. If *STARTANGLE* is 0, the starting point will be (*CENTERX (CENTERY + RADIUS)*). If *NDEGREES* is positive, the arc will be counterclockwise. If *NDEGREES* is negative, the arc will be clockwise. The other arguments are interpreted as described in **DRAWCIRCLE**.

Section 27.8 Miscellaneous Drawing and Printing Operations

(III:27.20)

To have a filled polygon print correctly, set the global variable **PRINTSERVICE** to floating point value 9.0 for printers running Services 9.0 or later.

When using **FILLPOLYGON** to be sent to Xerox 8044 Interpress printers, the global variable **PRINTSERVICE** must be set to the same value as the Print Service installed on your printer, currently either 8.0, 9.0 or 10.0. Thus, if your printer is running Print Service 9.0, you must set the global variable **PRINTSERVICE** to the

floating point value 9.0. This works around an incompatible change in the Xerox 8044 Interpress implementation.

The following function was omitted from previous version of the *Interlisp-D Reference Manual*:

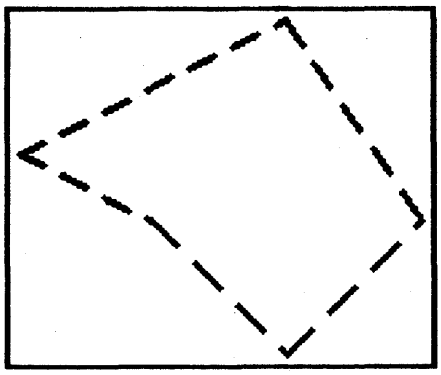
(DRAWPOLYGON *POINTS CLOSED BRUSH DASHING STREAM)* [Function]

Draws a polygon on the image stream *STREAM*. *POINTS* is a list of positions to which the figure will be fitted (the vertices of the polygon). If *CLOSED* is non-NIL, then the starting position is specified only once in *POINTS*. If *CLOSED* is NIL, then the starting vertex must be specified twice in *POINTS*. *BRUSH* and *DASHING* are interpreted as described in Chapter 27 of the *Interlisp-D Reference Manual*.

For example,

```
(DRAWPOLYGON '((100 . 100) (50 . 125)
               (150 . 175) (200 . 100) (150 . 50))
             T '(ROUND 3) '(4 2) XX)
```

would draw a polygon like the following on the display stream XX.



(III:27.20)

The function **FILLPOLYGON** contains two new arguments, *OPERATION* and *WINDNUMBER*. The new form for the function, and definitions for added arguments, follow.

(FILLPOLYGON *POINTS TEXTURE OPERATION WINDNUMBER STREAM)* [Function]

OPERATION is the **BITBLT** operation (see page 27.15 in the *Interlisp-D Reference Manual*) used to fill the polygon. If the *OPERATION* is **NIL**, the *OPERATION* defaults to the *STREAM* default *OPERATION*.

WINDNUMBER is the number for the winding rule convention . This number is either 0 or 1; 0 indicates the "zero" winding rule, 1 indicates the "odd" winding rule.

When filling a polygon, there is more than one way of dealing with the situation where two polygon sides intersect, or one polygon is fully inside the other. Currently, **FILLPOLYGON** to a display stream uses the "odd" winding rule, which means that intersecting polygon sides define areas that are filled or not filled somewhat like a checkerboard. For example,

```
(FILLPOLYGON '((125 . 125) (150 . 200) (175 . 125)
```

```
(125 . 175) (175 . 175))
GRAYSHADE WINDOW)
```

would produce a display something like this:

This fill convention also takes into account all polygons in *POINTS*, if it specifies multiple polygons.

Section 27.12 Fonts

A revised set of font printing metrics is a part of the Lyric release of Xerox Lisp. Note that Koto font files are still available to users who request them.

With the revised font set the interline spacing (line leading) is now consistent across all fonts within a point size. Previously, text with multiple fonts (but with the same point size, i.e., if a word were made bold or italic, or if the family were changed) would have different leading on different lines. The new .WD files clean up document appearance.

Note that these printer metric changes affect only hardcopy, not the display. The contents of the display fonts are essentially unchanged in Lyric.

Generally, line leading in the Lyric font files is tighter than in previous releases of the fonts. The default line leading is now the same as the font's nominal point size. As a consequence of the above, any text file (one not already formatted for Interpress) which is printed after installation of the new fonts will be formatted to a different length. This means that decisions regarding TEdit line leading, widows and orphans, left/right pages, references to page numbers, etc. will need to change. Koto documentation produced by users may need to be reformatted with different line leading, using the new fonts.

All of the font files now have a new naming scheme, which allows **FONTSAVAILABLE** to be able to do more accurate pattern matching. For example, the display font file for modern 8 bold italics used to be named:

```
Modern8-B-I-C41.Displayfont.
```

The file is now named:

```
Modern08-BIR-C41.Displayfont
```

In general font files use the following format:

The family name (e.g., `Modern`); a two digit size (e.g., 08); a three letter Face (e.g., `BIR`, for Bold Italic Regular); the letter C followed by the font's character set in base 8 (e.g., `C41`); and finally an extension (e.g., `Displayfont`).

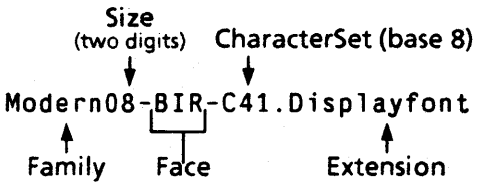


Figure 1. How the new font files are named. The three letter Face is composed of a weight (e.g., Bold), a slope, (e.g., Italic) and an expansion (e.g., Regular).

The old file naming convention is still supported, however, with the exception of the old Strike file naming convention. In Lyric, **FONTCREATE** will first search for fonts using the new font naming convention, and if the desired font is not found it will search using the Koto convention.

Compatibility considerations You can continue using the old printer metrics (.WD files) in Lyric, thus preserving document looks between Koto and Lyric. If you choose to do so, it is recommended that you rename your old .WD files to the new naming scheme (see above), so that you benefit from the changes to the font searching mechanisms. However, we strongly urge you to use the new .WD files. Otherwise, if you exchange TEdit documents with a site that is using the new files, the documents will print differently at the two sites. The creation date, rather than the naming convention, determines whether a .WD file represents the old or new format.

If, after installing the new .WD files, you wish to print a document using the old Koto formatting, make the font variable **INTERPRESSFONTDIRECTORIES** point to a directory containing the Koto font files. Also any Lyric printer font file information must be uncached from the sysout. To uncache the fonts, perform

```
(for INFO in (FONTSAVAILABLE '* '* '* '*
                    'INTERPRESS)
   do (APPLY 'SETFONTDESCRIPTOR INFO))
```

(III:27.30)

(STRINGWIDTH *STR FONT FLG RDTBL*) [Function]

In Lyric **STRINGWIDTH** observes ***PRINT-LEVEL*** and ***PRINT-LENGTH***.

Some new font manipulation functions have been added to Xerox Lisp. They are:

(WRITESTRIKEFONTFILE *FONT CHARSET FILENAME*) [Function]

Takes a display font fontdescriptor and a character set number, and writes that character set into a file suitable for reading in again. Note that the font descriptor's current state is used (which was perhaps modified by INSPECTing the datum), so this provides a mechanism for creating/modifying new fonts.

For example:

```
(WRITESTRIKEFONTFILE (FONTCREATE 'GACHA 10) 0
    '{DSK}Magic10-MRR-C0.DISPLAYFONT)
```

writes a font file which is identical in appearance to the current state of Gacha 10 charset 0.

If your DISPLAYFONTDIRECTORIES includes {DSK}, then a subsequent (FONTCREATE 'MAGIC 10) will create a new font descriptor who's appearance is the same as the old Gacha font descriptor.

However, the new font is identical to the old one in appearance only. The individual datatype fields and bitmap may not be the same as those in the old font descriptor, due to peculiarities of different font file formats.

Section 27.13 Font Files and Font Directories

(III:27.31)

Press fonts are not a part of the standard Xerox environment since PRESS is now a Library module.

Section 27.14 Font Classes

(III:27.32-27.48)

This section has been expunged from the *InterLisp-D Reference Manual*. Renumber the sections which followed the old Section 27.14 as

SECTION 27.15 ⇒ SECTION 27.14 Font Profiles

SECTION 27.16 ⇒ SECTION 27.15 Image Objects

SECTION 27.17 ⇒ SECTION 27.16 Implementation of Image Streams

Section 27.14 Font Profiles

(III:27.34)

The variable **FONTCHANGEFLG** has an additional value, **ALL**. **FONTCHANGEFLG = ALL** indicates that all calls to **CHANGEFONT** are executed.

(III:27.33)

The function **FONTNAME** is no longer supported by Interlisp-D.

Chapter 28 Windows and Menus

Section 28.1 Using the Window System

The default layout for the screen in the Lyric release has been altered from the Koto release. There is a new logo window (see figure 2).

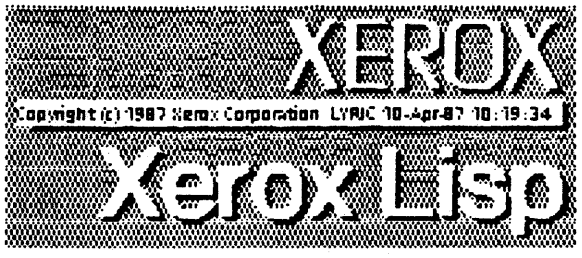


Figure 2. The Lyric Logo window

The default position for the logo window is the upper right corner of the screen.

Section 28.4 Windows

(III:28.13, 28.38)

The **ADDMENU** function will change a window's **RESHAPEFN** and also will change the window's **REPAINTFN**.

Section 28.4.5 Reshaping Windows

(III:28.17)

The Xerox Lisp window system allows the following minimum window sizes:

When creating a new window, the width and height specified must be at least 9, or else you will get an error "region too small to use as a window"

When reshaping a window, the smallest shape you can get is width = 26 and height = height of the font to be used in the window. If you specify a smaller region, **SHAPEW** will simply adjust it to fit these limits.

Section 28.4.8 Shrinking Windows Into Icons

(III:28.22)

SHRINKFN [Window property]

In previous releases, there was a bug in the attached window system such that if an attached window had a **SHRINKFN** of the single symbol DON'T, attempting to shrink the window resulted in a break with the message "UNDEFINED FUNCTION DON'T." For this case in Lyric, all windows that can be shrunk will be, while those windows with a **SHRINKFN** of the symbol DON'T will be left open.

To facilitate the management of window regions, the window property **EXPANDREGIONFN** has been added to Xerox Lisp. This feature allows applications to arrange for reshaping a window when it is expanded.

EXPANDREGIONFN [Window property]

EXPANDREGIONFN, if non-**NIL**, should be the function to be called (with the window as its argument) before the window is actually expanded.

The **EXPANDREGIONFN** must return **NIL** or a valid region, and must not do any window operations (e.g., redisplaying). If **NIL** is returned, the window is expanded normally, as if the **EXPANDREGIONFN** had not existed. The region returned specifies the new region for the main window only, not for the group including any of its attached windows. The window will be opened in its new shape, and any attached windows will be repositioned or rejustified appropriately. The main window must have a **REPAINTFN** which can repaint the entire window under these conditions.

As with expanding windows normally, the **OPENFN** for the main window is not called.

Also, the window is reshaped without checking for a special shape function (e.g., a **DOSHAPEFN**).

(III:28.23)

Add the variable **DEFAULTICONFN** to the Icon section of the *InterLisp-D Reference Manual*:

DEFAULTICONFN [Variable]

Changes how an icon is created when a window having no **ICONFN** is shrunk or when **SHRINKW,** with a *TOWHAT* argument of a string, is called. The value of **DEFAULTICONFN** is a function of two arguments (window text); text is either **NIL** or a string. **DEFAULTICONFN** returns an icon window.

The initial value of **DEFAULTICONFN** is **MAKETITLEBARICON**. It creates a window that is a title bar only; the title is either the text argument, the window's title, or "Icon made <date>" for titleless windows. **MAKETITLEBARICON** places the title bar at some corner of the main window.

An alternative behavior is available by setting **DEFAULTICONFN** to be **TEXTICON**. **TEXTICON** creates a titled icon window from the text or window's title. It is described further in Appendix B (ICONW).

Section 28.4.11 Terminal I/O and Page Holding

(III:28.29)

TTYDISPLAYSTREAM has been fixed so that it can be successfully used with non-windows.

Section 28.5 Menus

Two features have been added to this section, ICONW for creating icons, and FREE MENU, for creating and using free menus. Both features were formerly part of the Lisp Library.

The description for ICONW is in Appendix C. The FREE MENU description is in Appendix D.

The Lyric version of Free Menu differs in some respects from the Koto version of Free Menu. Following is a description of the incompatible feature changes from the old version to the new

version of Free Menu. Some of the terminology used in these notes is introduced in the Free Menu documentation found in Appendix B. Please reference Appendix B before reading the following notes.

- The function **FREEMENU** is used to create a Free Menu, replacing and combining the functions **FM.MAKEMENU** and **FM.FORMATMENU**.

The description of Free Menu has these changes:

1. There is no longer a **WINDOWPROPS** list in the Free Menu Description. Instead, the window properties **TITLE** and **BORDER** thatwere previously set in the **WINDOWPROPS** list can now be passed to the function **FREEMENU**. Other window properties (like **FM.PROMPTWINDOW**) can be set directly after Free Menu returns the window using the system function **WINDOWPROP**. See Appendix B, Section 28.7.14, Free Menu Window Properties.

2. Setting the initial state of an item is now done with the item property **INITSTATE** in the item description, rather than the **STATE** property.

Free Menu Items has been modified as follows:

1. **3STATE** items now have states **OFF**, **NIL**, and **T** (instead of a **NEUTRAL** state). They appear by default in the **NIL** state.

2. **STATE** items are general purpose items which maintain state, and replace the functionality of **NCHOOSE** items. To get the functionality of **NCHOOSE** items, specify the property **MENUITEMS** (a list of items to go in a popup menu), which instructs the **STATE** item to popup the menu when it is selected. **STATE** items do not display their current state by default, like **NCHOOSE** items used to. Instead, if you want the state displayed in the Free Menu, you have to link the **STATE** item to a **DISPLAY** item using a Free Menu Item Link named "DISPLAY". The current state of the **STATE** item will then automatically be displayed in the specified **DISPLAY** item. The item properties **MENUFONT** and **MENUTITLE** also apply to the popup menu.

3. **NWAY** items are declared slightly differently. There is now the notion of an NWay Collection, which is a collection of items acting an a single nway item. The Collection is declared by specifying any number of NWay items, each with the same **COLLECTION** property. NWay Collections have properties themselves, accessible by the macro **FM.NWAYPROPS**. These properties can be specified in property list format as the value of the **NWAYPROPS** Item Property of the first NWay item declared for each Collection. NWay Collections by default cannot be deselected (a state in which no item selected). Setting the Collection property **DESELECT** to any non-nil value changes this behavior. The state of the NWay Collection is maintained in its **STATE** property.

4. **EDIT** items no longer will stop at the edge of the window. Editing is either restricted by the **MAXWIDTH** property, or

else it is not restricted at all. The **EDITSTOP** property is obsolete. When you start editing with the right mouse button the item is first cleared.

5. **EDITSTART** items now specify their associated edit item (there can only be one, now) by a Free Menu Item Link named "EDIT" from the **EDITSTART** item to the **EDIT** item.

6. **TITLE** items are replaced by **DISPLAY** items, which work the same way.

With Free Menu, the item interface functions can take the actual item datatype, the item's *ID* or *LABEL*, or a list of the form (**GROUPID** *ITEMID*) specifying a particular item in a group, as the *ITEM* argument.

The description for ICONW is in Appendix B. The FREE MENU description is in Appendix C.

These changes have occurred in the Free Menu Interface functions:

(FREEMENU *DESCRIPTION TITLE BACKGROUND BORDER*) [Function]

replaces **FM.MAKEMENU** and **FM.FORMATMENU**. The desired format is not specified as the value of the **FORMAT** property in the group's PROPS list.

(FM.GETITEM *ID GROUP WINDOW*) [Function]

replaces **FM.ITEMFROMID**.

Searches within *GROUP* for an item whose ID property is *ID*.

ID is matched against the item ID and then the item **LABEL**. If *GROUP* is **NIL**, the entire menu is searched.

(FM.GETSTATE *WINDOW*) [Function]

replaces **FM.READSTATE**.

Returns a property list of the selected item in the menu. This list now also includes the NWay Collections and their selected item.

(FM.CHANGELABEL *ITEM NEWLABEL WINDOW UPDATEFLG*) [Function]

has a new argument order. Now works by rebuilding the item label from scratch, taking the original specification of **MAXWIDTH** and **MAXHEIGHT** into account. *NEWLABEL* can be an atom, string, or bitmap. If *UPDATEFLG* is set, then the Free Menu Group's regions are recalculated, so that boxed groups will be redisplayed properly.

(FM.CHANGESTATE *X NEWSTATE WINDOW*) [Function]

has a new argument order.

X is either an item or an NWay Collection ID. *NEWSTATE* is an appropriate state to the type of item. If an NWay collection, *NEWSTATE* is the actual item to be selected, or **NIL** to deselect. Toggle items take either **T** or **NIL** as NEWSTATE, and **3STATE** items take **OFF**, **NIL**, or **T**, and **STATE** items take any atom, string, or bitmap as their new state. For **EDIT** items, *NEWSTATE* is the new label, and **FM.CHANGELABEL** is called to change the label of the **EDIT** item.

(**FM.RESETSHAPE** *WINDOW ALWAYSFLG*) [Function]

replaces **FM.FIXSHAPE**

(**FM.HIGHLIGHTITEM** *ITEM WINDOW*) [Function]

replaces **FM.SHADEITEM** and **FM.SHADEITEMBM**.

FM.HIGHLIGHTITEM will programmatically highlight an item, as specified by its **HIGHLIGHT** property. The highlighting is temporary, and will be undone by a redisplay or scroll. To programmatically shade an item an arbitrary shade, use the new function **FM.SHADE**.

Section 28.6.2 Attached Prompt Windows

(**GETPROMPTWINDOW** *MAINWINDOW #LINES FONT DONTCREATE* [Function]

In the Lyric release, the prompt window created by **GETPROMPTWINDOW** is *not* independently closeable, as it was in Koto. That is, selecting **Close** from the right-button window menu in the prompt window is the same as selecting it from the menu of any other window in the group—the entire window group is closed.

Chapter 29 Hardcopy Facilities

(III:29.3)

The **HARDCOPYW** function now has an additional argument, *HARDCOPYTITLE*, which allows you to change or eliminate the "Window Image" message on IP screen images. Moreover, **HARDCOPYW** function now allows you to print large images occupying more than one page.

(**HARDCOPYW** *WINDOW/BITMAP/REGION FILE HOST SCALEFACTOR ROTATION PRINTERTYPE HARDCOPYTITLE*) [Function]

HARDCOPYTITLE is a string specifying a title to print on the page containing the screen image. If NIL, the string "Window Image" is used. To omit a title, specify the null string.

Chapter 30 Terminal Input/Output

Section 30.1 Interrupt Characters

(III:30.2)

Control-P The Control-P (**PRINTLEVEL**) interrupt is no longer supported. The interrupt of that name still exists and is defaultly assigned to Control-P, but has no effect on printing.

Control-T — The Control-T interrupt flashes the window belonging to the tty process and prints its status information in the prompt window. This avoids disrupting the user typescript.

(III:30.3)

(**INTERRUPTCHAR** CHAR TYP/FORM HARDFLG —) [Function]

If the argument *TYP/FORM* is a symbol designating a predefined system interrupt (**RESET**, **ERROR**, **BREAK**, etc), and *HARDFLG* is omitted or **NIL**, then the hardness defaults to the standard hardness of the system interrupt (e.g., **MOUSE** for the **ERROR** interrupt).

Section 30.2.3 Line Buffering

(III:30.11-12)

The **BKSYSBUF** function has been changed, for compatibility reasons. The description now reads as follows:

(**BKSYSBUF** *X FLG RDTBL*) [Function]

BKSYSBUF appends the **PRIN1**-name of *X* to the system input buffer. The effect is the same as though the user had typed *X*. Returns *X*.

If *FLG* is *T*, then the **PRIN2**-name of *X* is used, computed with respect to the readtable *RDTBL*. If *RDTBL* is **NIL** or omitted, the current readtable of the TTY process (which is to receive the characters) is used. Use this for copy selection functions that want their output to be a readable expression in an Exec.

Note that if you are typing at the same time as the **BKSYSBUF** is being performed, the relative order of the typein and the characters of *X* is unpredictable.

(III:30.12)

Add the function **BKSYSCHARCODE** used in line buffering:

(**BKSYSCHARCODE** *CODE*) [Function]

This function appends the character code *CODE* to the system input buffer. The function **BKSYSBUF** is implemented by repeated calls to **BKSYSCHARCODE**.

Section 30.4.1 Changing the Cursor Image

(III:30.14)

The **CURSOR** record has been changed to a DATATYPE, and its field names have changed in the following way:

Old Field Name	New Field Name
CURSORBITMAP	**CUIMAGE**
CURSORHOTSPOTX	**CUHOTSPOTX**
CURSORHOTSPOTY	**CUHOTSPOTY**

The **CURSORHOTSPOT** field no longer exists; its value can be fetched by composing **CUHOTSPOTX** and **CUHOTSPOTY** into a

POSITION, or stored by destructuring a **POSITION** into those fields.

Section 30.5 Keyboard Interpretation

(III:30. 19–20)

(**KEYDOWNP** *KEYNAME*) [Function]

(**KEYACTION** *KEYNAME ACTIONS* —) [Function]

KEYNAME is interpreted differently in Lyric: If *KEYNAME* is a small integer, it is taken to be the *internal* key number. Otherwise, it is taken to be the name of the key. This means, for example, that the name of the "6" key is not the number 6. Instead, spelled-out names for all the digit keys have been assigned. The "6" key is named SIX. It happens that the key number of the "6" key is 2. Therefore, the following two forms are equivalent:

```
(KEYDOWNP 'SIX)

(KEYDOWNP 2)
```

Note: The key labeled HELP on the 1186 is named DBK-HELP for use in KEYACTION.

Section 30.6 Display Screen

(III:30.22-23)

(**CHANGEBACKGROUND** SHADE —) [Function]

The function **CHANGEBACKGROUND** treats the *SHADE* argument as a 4 X 4 texture. The **CHANGEBACKGROUNDBORDER** function, on the other hand, treats the *SHADE* argument as a 2 X 8 texture.

Therefore, note that the same *SHADE* argument, when used by the two functions, will not necessarily produce the same background and border shades on the display screen.

(III:30.23)

The **VIDEORATE** function works only on the 1108. Append the following note to the **VIDEORATE** function description:

(**VIDEORATE** TYPE) [Function]

Note: **VIDEORATE** does not work on the 1186.

Section 30.7 Miscellaneous Terminal I/O

(III:30.24)

(**BEEPON** FREQ) [Function]

The argument *FREQ* is measured in hertz, not in **TICKS**.

Chapter 31 Ethernet

Section 31.3.1 Name and Address Conventions

(III:31.8-9)

Amend the first paragraph, describing **NSADDRESS**, to list, in order, the components of **NSADDRESS**:

Addresses of hosts in the NS world consist of three parts, a network number, a machine number, and a socket number. These three parts are embodied in the Interlisp-D data type **NSADDRESS**. The components of **NSADDRESS** are 32-bit network, 48-bit host, 16-bit socket.

Move the following sentence from page 31.9 to the last paragraph of Name and Address Conventions on page 31.8:

If you wish to manipulate **NSADDRESS** and **NSNAME** objects directly you should load the Lisp Library Module ETHERRECORDS.

Section 31.3.2 Clearinghouse Functions

(III:31.9)

The variable **AUTHENTICATION.NET.HINT** has been added to Clearinghouse Functions. It follows the **CH.NET.HINT** variable in the *Interlisp-D Reference Manual*.

AUTHENTICATION.NET.HINT [Variable]

AUTHENTICATION.NET.HINT can be set to **CH.NET.HINT** to speed up the initial authentication connection. Its value is interpreted in the same manner as **CH.NET.HINT**.

Section 31.3.5.3 Performing Courier Transactions

(III:31.20-21)

The **COURIER.OPEN** function requires that a courier server be running on the host machine.

Section 31.5 Pup Level One Functions

\10MBTYPE.PUP [Variable]

\10MBTYPE.3TO10 [Variable]

The values of these variables are the 10MB Ethernet encapsulation types for PUP packets and Pup-to-10MB address translation packets, respectively. The initial values of these variables are 512 and 513, respectively. However, these values are illegal for an Ethernet conforming to IEEE 802.3 specifications.

New encapsulation types have been defined for IEEE 802.3 networks. To use them, set the variable **\10MBTYPE.PUP** to 2560 (decimal) and **\10MBTYPE.3TO10** to 2561. Then call either **(RESTART.ETHER)** or **(LOGOUT)**, so that the Ethernet code can reinitialize itself. It may be convenient for a site to smash these values directly into the standard sysout everyone fetches by using

the function **READSYS** and its ↑V command from the TeleRaid Library module (the sysout must be on disk or a random-access file server). Note that *all* pup hosts on a network (servers as well as workstations) must simultaneously choose to use the new values; those using different values will be unable to communicate with each other. The System Tool must also be upgraded at the same time.

Section 31.6.1 Creating and Managing XIPs

The function NSNET.DISTANCE was previously undocumented. The documentation is:

(NSNET.DISTANCE *NET#***)** [Function]

Returns the "hop count" to network *NET#*, i.e., the number of gateways through which an XIP must pass to reach *NET#*, according to the best routing information known at this point. The local (directly-connected) network is considered to be zero hops away. Current convention is that an inaccessible network is 16 hops away. **NSNET.DISTANCE** may need to wait to obtain routing information from an Internetwork Router if *NET#* is not currently in its routing cache.

5. LIBRARY MODULES

Since the Koto release of this manual, the following changes have taken place:

The name was changed, from "packages" to "modules," because the former term has a specific meaning in Common Lisp.

Several modules were taken out of the library, and put into LispUsers.

At the same time, four modules that were part of the Interlisp environment have now been placed into the library.

And all modules presently in the library have been reviewed and edited both for technical changes and style of presentation.

Modules Moved from the Library to LispUsers

Big
BitMapFns
BusExtender
BusMaster
CirclPrint
CheckSet
CompileBang
Color
C150Stream
DECL
DInfo
FileCache
HelpSys
Iris
LambdaTran
PCallStats
ReadAIS

Modules Moved to Their Own Manuals

TEdit
Sketch
CML, CMLArray, CMLArrayInspector (part of Xerox Common Lisp)

Modules Moved From the Sysout Into the Library

DEdit
Masterscope
Match
Press

Modules Moved From the Library Into the Sysout

IconW
FreeMenu

Modules Replaced

Old: FX-80stream, FastFX-80stream, FXprinter
New: FX-80Printer

New Modules

SysEdit
TableBrowser

New Features Since KOTO

The following list is meant to indicate the highlights, rather than be a complete and exhaustive summary of all the new features that were added and other changes that were made since the Koto release.

4045XLPstream	Enabled its graphics capabilities; added 1108 cable/connector pin-outs.
Centronics	Added cable/connector pin-out.
Chat	Added information about EMACS.
CopyFiles	When told to copy to a non-existent NS subdirectory, it now asks if it should create it.
EditBitMap	Added a description of its user interface.
FileBrowser	Added enhanced features to Load, Compile, Edit; it now preserves path name of source files when copying to another machine or user; sorts files by attributes; and prints hard copies of directory listings.
FX-80driver	New software, new text; added 1108/1186 cable/connector pin-outs.
Kermit	Added reference to a really nice text/reference book.
NSmaintain	Clarified the command set.
TCP-IP	Added revised/expanded installation procedure.
TExec	Clarified its purpose in life.

Additional Notes

DEdit is not error-protected. Doing a ↑ in a DEdit break window closes the DEdit window, too...

In addition, the modules work under all Xerox Lisp environments (Interlisp-D, Common Lisp, Xerox Lisp). However, many of the functions and variables used within the modules are those of Interlisp-D, and therefore you'll have to make sure that, when you are not in Interlisp, you use the IL: prefix (see the *Release Notes* for more details).

Koto CML Library Module

If you have files that used the Koto CML library module, with its package-style symbol naming conventions, you will need to

convert them to use the correct symbols in Lyric. The procedure is briefly as follows; see the *Xerox Comon Lisp Implementation Notes*, Chapter 11, "Reader compatability feature" for complete details on this mechanism:

First, set the global variable LITATOM-PACKAGE-CONVERSION-ENABLED to T. Then for each of your files, do

(LOAD *file* 'PROP)

(MAKEFILE *file* 'NEW)

Afterwards be sure to set the global variable LITATOM-PACKAGE-CONVERSION-ENABLED back to NIL.

[This page intentionally left blank]

A User's Guide to TEdit—Release Notes

Changes, Additions, Corrections to TEdit Part One

TEdit now accepts non-TEdit documents that contain extended NS characters. It no longer shows as characters interspersed with black boxes (AR 4545).

TEdit now accepts non-TEdit documents that contain extended NS characters. It no longer shows as characters interspersed with black boxes (AR 4545).

Paragraph Looks Menu

If you apply "new-page-before" to the first paragraph on the first page, TEdit will no longer skip a page when the file is printed.

Page Layout Menu

You may now specify a landscape page layout.

In the page layout menu, Modern 10 MRR is now the default page number font instead of Gacha 10. Also, there is a global variable, TEDIT.DEFAULT.FOLIO.LOOKS, that you can set to be any character-looks specification acceptable to TEDIT.LOOKS. The default (i.e., if you don't specify one in the page layout menu) is taken from there.

If you have set page formatting in the past, the page-numbering font has been set as well (even if you specified nothing). This behavior continues, but the default is more sensible, and can be changed.

You may now number the first page of a TEdit file 0(zero).

TEdit now preserves text before and after page numbers after a file is saved.

Using numbers with decimal points in the "Text before page number" field in the page-layout menu now works properly.

New Features

TEdit GET now offers you the current/last-typed file name in the same manner as TEdit PUT.

The copyright symbol has been added to TEdit abbreviations (NS fonts only, e.g., Modern, Classic, etc.). If you expand a lowercase c you will get the copyright symbol.

Changes, Additions, Corrections to Modifying TEdit

Note: This section is now called the Programmer's Interface to TEdit

STREAM AND TEXTOBJ

All public TEdit functions (non- \) that take a *TEXTOBJ* argument accept either a TEXTOBJ or a text STREAM as that argument's value.

Changes, Additions and Corrections to TEdit functions

The function TEDIT.SINGLE.PAGEFORMAT is incorrectly documented in the Lisp Library. The following corrections should be noted: The arguments *PG#X*, *PG#Y*, and *PG#FONT* should be *PX*, *PY*, and *PFONT*, respectively.

The argument *PG#ALIGNMENT* should be *PQUAD*.

The order for the arguments, *TOP BOTTOM LEFT RIGHT* should be *LEFT RIGHT TOP BOTTOM*.

The argument *#COLS* should be *COLS*.

INTERCOLSPACE should be *INTERCOL*. And between the *INTERCOL* and *UNITS* arguments there is a *HEADINGS* argument.

The functions and its arguments look like:

(TEDIT.SINGLE.PAGEFORMAT *PAGE#S? PX PY PFONT PQUAD LEFT RIGHT TOP BOTTOM COLS COLWIDTH INTERCOL HEADINGS UNITS PAGEPROPS PAPERSIZE*) [Function]

PAGE#S? T if you want page numbers on this kind of page, else NIL.

PX The horizontal location of the page number, measured from the left edge of the paper. Negative values are measured from the paper's right edge.

PY The vertical location of the baseline for the page numbers, measured from the bottom of the paper. Negative values are measured from the top of the paper.

PFONT The font to be used to display the page numbers. This can be any specification that is acceptable to TEDIT.LOOKS.

PQUAD An atom that tells how the page number is to be aligned on the location specified by *PX* and *PY*. LEFT means the location is the lower-left corner of the page number. RIGHT means the location is the lower-right corner. CENTERED means the page number will be centered around the *PX* you specified.

LEFT The left margin—the distance from the left edge of the paper to the left edge of the first text column.

RIGHT The right margin—the distance from the right edge of the rightmost text column to the right edge of the paper.

TOP The top margin of the page—the distance from the top of the paper to the top of the first line of body text.

BOTTOM The bottom margin—the distance from the bottom of the last line of body text to the bottom of the paper.

COLS Number of columns (default is one).

COLWIDTH The column width (default is to evenly divide the available space among the *#COLS* columns).

INTERCOL The space between the right edge of one column and the left edge of the next column. Defaults to evenly divide the space left after the columns are set up. If there is more than one column, one or the other of *COLWIDTH* and *INTERCOLSPACE must* be specified.

HEADINGS A list of lists in the form of ((HEADINGNAME$_1$ XLOCATION$_1$ YLOCATION$_1$) (HEADINGNAME$_2$ XLOCATION$_2$ YLOCATION$_2$) . . .(HEADINGNAME$_n$ XLOCATION$_n$ YLOCATION$_n$)).

UNITS The units used in setting the values you specified. May be one of the atoms PICAS, IN, INCHES, CM, POINTS. Default is POINTS.

PAGEPROPS A property list of extra information. Properties are STARTINGPAGE#, FOLIOINFO, and LANDSCAPE?.

STARTINGPAGE# is the first page's number; it is ignored if this isn't the first page.

FOLIOINFO is a list of information about page numbers, (FORMAT TEXTBEFORE TEXTAFTER). FORMAT can be one of ARABIC, LOWERROMAN, UPPERROMAN, or NIL (i.e., ARABIC). TEXTBEFORE is the text preceding the number, and TEXTAFTER is the text following the number.

LANDSCAPE? determines if the document is printed in the usual vertical format or printed in landscape format (horizontally). If NIL the document is printed vertically, if non-NIL the document is printed landscape. Defaults to NIL.

PAPERSIZE Is one of LETTER, LEGAL, the metric paper sizes (A0, A1, A2 A3, A4, A5, B0, B2, B3, B4), or NIL (which defaults to letter size).

TEDIT.GET accepts an open stream as the file to GET from. You may still pass it a TEXTOBJ, however.

(TEDIT.GET *STREAM FILE UNFORMATTED?*) [Function]

Performs the TEdit Get command, loading the text from *FILE* onto the editing stream *STREAM*—replacing the text that is being edited currently. If *FILE* is not supplied, the user will be asked for a file name. If *UNFORMATTED?* is non-NIL, *FILE* is treated as a plain-text document, and all of its contents are included—even TEdit formatting information.

You can now use TEDIT.PUT to store a TEdit document in the middle of a larger file (e.g., for saving TEdit documents as part of a database). The complete documentation is now as follows:

(TEDIT.PUT *STREAM FILE FORCENEW UNFORMATTED? OLDFORMAT?*) [Function]

Performs the TEdit Put command, saving the text from the text stream *STREAM* onto the file named *FILE*. If *FILE* is NIL, the user will be prompted for a file name. In this case, if *FORCENEW* is

NIL, the user is offered the old file name as a default; if non-NIL, no default is given, forcing the user to specify a file name. If *UNFORMATTED?* is non-NIL, only characters are put in the file—no formatting. If *OLDFORMAT?* is non-NIL, the file will be written in the format used by the previous version of TEdit, for backward compatibility.

In order to store a TEdit document as part of another file, call TEDIT.PUT, passing an open stream on the file as the *FILE* argument. The stream should be open for output and positioned at the place you want TEdit to store the document (call this file pointer START). When TEDIT.PUT returns, the stream's end-of-file pointer will be just after the last byte in the newly-inserted document. Call this file pointer END. To subsequently retrieve the document from the middle of this other file, call OPENTEXTSTREAM on the file, passing the START and END pointers as the *START* and *END* arguments.

Note: When TEDIT.PUT returns, the stream will be open for INPUT.

The functions TEDIT.MOVE and TEDIT.COPY were not documented in Koto. They are:

(TEDIT.MOVE *FROM TO*) [Function]

FROM and *TO* are SELECTIONs. Moves the text described by *FROM* to the place described by *TO*, within the same text stream or between different text streams. The text described by *FROM* is deleted from its original location.

(TEDIT.COPY *FROM TO*) [Function]

FROM and *TO* are SELECTIONs. Copies the text described by *FROM* to the place described by *TO*, within the same text stream or between different text streams. The text described by *FROM* is not deleted in the *FROM* location.

Changes in the Documentation of TEdit Functions

The following functions have had the documentation of their arguments changed to reflect what will appear if you do a ?= or evaluate ARGLIST on one of these functions. Arguments that were corrected are indicated by bold italics (***arg***). Please note that what changed was the documentation, not the way the functions operate or the values of the arguments themselves.

(TEDIT.SETSEL *STREAM **CH#** LEN POINT **PENDINGDELFLG** LEAVECARETLOOKS OPERATION*) [Function]

(COERCETEXTOBJ *STREAM TYPE **OUTPUTSTREAM***) [Function]

(TEDIT.DELETE *STREAM **SEL** LEN*) [Function]

(TEDIT.INCLUDE ***STREAM** FILE START END*) [Function]

(TEDIT.FIND ***STREAM TARGETSTRING** START# END# WILDCARDS?*) [Function]

(TEDIT.GET.LOOKS ***STREAM CH#ORCHARLOOKS***) [Function]

(TEDIT.PARALOOKS ***STREAM** NEWLOOKS **SEL LEN***) [Function]

(TEDIT.COMPOUND.PAGEFORMAT ***FIRST VERSO RECTO***) [Function]

(TEXTOBJ ***STREAM***) [Function]

(TEXTSTREAM *STREAM*) [Function]
(TEDIT.CARETLOOKS *STREAM LOOKS*) [Function]
(TEDIT.NORMALIZECARET *STREAM SEL*) [Function]
(COPYTEXTSTREAM *ORIGINAL CROSSCOPY*) [Function]
(TEDIT.PROMPTPRINT *TEXTSTREAM MSG CLEAR?*) [Function]
(TEDIT.SETSYNTAX *CHAR CLASS TABLE*) [Function]
(TEDIT.GETSYNTAX *CH TABLE*) [Function]
(TEDIT.SETFUNCTION *CHARCODE FN RTBL*) [Function]
(TEDIT.WORDGET *CH TABLE*) [Function]
(TEDIT.WORDSET *CHARCODE CLASS TABLE*) [Function]
(TEDIT.INSERT.OBJECT *OBJECT STREAM CH#*) [Function]

The following functions were previously documented as accepting a TEXTOBJ. They all still take a TEXTOBJ but they will now also accept a STREAM as the first argument.

(TEDIT.FIND *STREAM TARGETSTRING START# END# WILDCARDS?*) [Function]
(TEDIT.GET.LOOKS *STREAM CH#ORCHARLOOKS*) [Function]
(TEDIT.PARALOOKS *STREAM NEWLOOKS SEL LEN*) [Function]
(TEXTSTREAM *STREAM*) [Function]
(TEDIT.NORMALIZECARET *STREAM SEL*) [Function]
(TEDIT.PROMPTPRINT *TEXTSTREAM MSG CLEAR?*) [Function]

New Features

For the benefit of NS file server users, TEdit now writes files of type TEDIT, instead of BINARY. As a result, LISTFILES and the FileBrowser are able to determine that the file is a TEdit file and call TEdit to create the hardcopy. Previously, it was necessary that the TEdit file explicitly have the extension ".TEdit".

```
(OPENSTREAM file 'OUTPUT 'NEW '((TYPE TEDIT))).
```

This change is for formatted files only. Plain text files are still written as type TEXT. Also, on devices that don't support arbitrary file types (such as conventional mainframe file servers), the type TEDIT coerces to BINARY. Unfortunately, if you subsequently copy the file to an NS file server from such a device, the knowledge of its "true" file type is lost.

Fixed ARS

AR 883—You no longer get spurious dashed underlining if you scroll during a copy-selection and then extend the that selection from off-screen to on.

AR 4148—`(COERCETEXTOBJ textobj 'STREAM)` now returns a stream.

AR 5092—Splitting and unsplitting TEdit window is now much more robust.

AR 5093—TEdit Find and Substitute no longer ignore leading zeros.

AR 5539—Files on hosts that don't use CR as their end-of-line marker will not cause OPENTEXTSTREAM to break with an "end of file" error.

AR 5621—TEDIT.PUT no longer passes NIL to the PUTFN as the file name for a new file.

AR 5830—Programmatically Closing a TEdit window opened with OPENTEXTSTREAM works.

AR 5903—Calling TEDIT.PUT from the QUITFN now closes the old stream.

AR 5913—Using TEDIT.INSERT with pending delete selection no longer causes display to be inconsistent.

AR 5920—Selecting a bitmap (shift-select, delete-select, etc.) in a TEdit document will not cause the bitmap editor menu to appear.

AR 5933—Using decimal tabs no longer requires an extra character after a number to align properly.

AR 6088—When text is moved out of bounds of the window by control shift selection, TEdit redisplays correctly.

AR 6274—Koto TEdit hang if you tried to hardcopy a series of paragraphs that was bigger than a page with the HEADINGKEEP property set to ON. TEdit no longer hangs.

AR 6447—TEdit will not break when an attempt is make to PUT an empty file.

AR 6791—In Koto the function TEDIT.PUT.PCTB did not preserve NS characters if they immediately followed a bitmap. TEDIT.PUT.PCTB now preseves NS characters in all cases.

AR 6802—\PEEKBIN now works properly with image objects. \PEEKBIN no longer advances the fileptr.

A User's Guide to Sketch—Release Notes

The Lyric release of Sketch includes several new features, many added in response to user's requests. The Lyric version of Sketch also supports a programmer's interface which allows sketches to be created by programs. This interface is described in a separate document (*The Programmer's Interface to Sketch.*)

Manipulating Sketch Elements

Adding and Deleting Control Points

Individual control points can now be added to and deleted from wires and curves.

Deleting Control Points

You now have the option to delete elements or delete a control point. Just select the **Delete** command, move the mouse cursor out through the grey arrow, then select the point to be deleted.

Defaults Command

Better Feedback for Creating Wires, Circles and Ellipses

Sketch now provides better feedback when you are creating circles, ellipses and wires. You are now prompted with an image of what the figure will look like if you release the left button. You can get the old feedback behavior (for example, if this is too slow) by selecting the **Feedback** subcommand from the **Defaults** submenu, then selecting the **Points only** subcommand from its submenu.

Arrowheads

A curved arrowhead shape was added and is now the default. Also, a command was added to the menu of arrowhead change operations that implements "look same" for arrowheads. To make the arrowheads on a collection of elements look the same: select **Change**; then, when prompted to select the elements to change, first select the element that has the desired arrowhead, then, in the same selection, add the elements that you want to look like the first one; then select the item **Arrowheads**, then the item **Both**, then the item **Same as First**.

Deleting Characters During Type-in

You can now delete characters by using the UNDO key, just as you would in TEdit. Type in a word or a phrase, then press the UNDO key, and the text will be deleted.

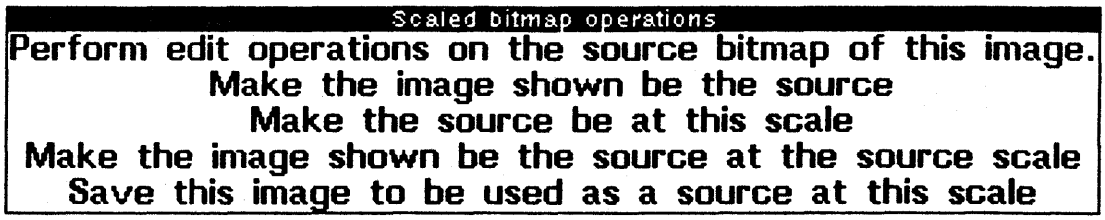
Using Bit Maps in a Sketch

Zooming Bitmaps

The bit image element provides a bitmap that zooms. Selecting the **Bit image** command from the command menu will prompt you for a region of the screen that will be inserted as a bit image into the sketch.

Changing Bitmaps

When you apply a **Change** command to a bit image that it is being viewed at actual size, you will be prompted with the same menu as a bitmap image object. If the image is being displayed at other than original scale, you will be given the menu shown below.



Menu of commands offered when the Change command is applied to a bit image that is not at original scale.

Freezing Sketch Elements

It is now possible to freeze elements, that is to make them unaffected by edit changes. Frozen elements will not have their control points highlighted (and hence cannot be selected) after an edit command has been selected. This provides a way to keep part of the figure fixed while editing on an overlapping part. It also reduces the number of control points. The **Freeze** command is a subcommand to the **Group** command. It will prompt you for a collection of elements that will then be frozen. Elements can be unfrozen by the **UnFreeze** command that is a subcommand to the **UnGroup** command.

Aligning Sketch Elements

Sketch contains a set of commands to align elements. The main menu command **Align** prompts for a collection of control points and moves them so that they all line up with the leftmost one.

Placing Multiple Copies of Elements

There is a new feature in Sketch that makes it much easier to place multiple copies of a collection of elements. While positioning the image of the elements during the **Copy** command, hold down the COPY key. A new copy of the elements will be positioned everytime a mouse button (left or right) is pressed and released, until either the image is placed completely outside the viewer or the COPY key is released before the mouse button is released.

Making the Window Fit the Sketch

The **Fit to window** subcommand under the **Move View** command will zoom the sketch so that it just fits within the current window. It has a sub-subcommand **Fit window to sketch** that will reshape the window so that the entire sketch (at the size shown) just fits within it. This is useful if you change a sketch that was edited from a document.

Overlaying Figure Elements

Elements that have a filling property (boxes, text boxes, circles, polygons and closed curves) now have a mode property that determines how the filling should effect elements it covers. The option **Filling mode** now appears in the **Which aspect?** submenu.

Changing How Elements Overlap

Elements have an order in which they are displayed. An element that is displayed early can be covered by elements layed down later. Thus, changing the order in which overlapping elements are displayed can effect the resulting image. The **Bury** command provides three subcommands to change the order in which elements are displayed.

The **Bury** command will prompt you to select an element or elements and will change their order so that they are displayed first. That is, they will appear underneath any other elements. If you select more than one element, they will all be displayed before any non-selected elements and their relative order maintained. The **Send to bottom** subcommand does the same thing as **Bury**.

The **Bring to top** command is a subitem to the **Bury** command. It will prompt you to select an element or elements and will change their order so that they are displayed last. That is, they will appear on top of any other elements. If you select more than one element, they will all be displayed after any non-selected elements and their relative order maintained.

The **Reverse order** command is a subitem to the **Bury** command. It will prompt you to select a collection of elements and will reverse their display orders. A special case is when two elements are selected. In this case the element positions are switched.

The Programmer's Interface

Since the Koto release, the programmer's interface to Sketch has been significant redesigned. The programmer's interface allows Sketch to be used as a tool by other programs. It is documented in t*he Programmer's Interface to Sketch*.

New Behavior for the Get Command

The action of the **Get** command was changed to be consistent with the TEdit **Get** command. It now deletes any sketch elements that are in the sketch prior to the **Get** command. The affect of the old **Get** command is available as the **Include** command on a submenu to the **Get** command.

Establishing Initial Defaults for Sketch

The variable SK.DEFAULT.FONT, if non-NIL, is used as the default font. If SK.DEFAULT.FONT is NIL, the default font (DEFAULTFONT) is used.

The following variables are used to establish the default setting for a new sketch. Descriptions of legal values can be found in the *Programmer's Interface to Sketch*. SK.DEFAULT.BRUSH is the default brush. SK.DEFAULT.ARROW.LENGTH is the default arrowhead size. SK.DEFAULT.ARROW.TYPE is the default type (one of LINE, CURVE, CLOSEDLINE or SOLID). SK.DEFAULT.ARROW.ANGLE is the default angle for arrowheads. SK.DEFAULT.TEXT.ALIGNMENT is the default text alignment. SK.DEFAULT.TEXTBOX.ALIGNMENT is the default textbox alignment. SK.DEFAULT.DASHING is the default dashing. SK.DEFAULT.TEXTURE, SK.DEFAULT.BACKCOLOR and SK.DEFAULT.OPERATION are combined to create the default filling.

1108 User's Guide Release Notes

What to Look For

The *1108 User's Guide* has been extensively reorganized and rewritten for the Lyric Release. Wherever possible, it is now nearly identical to the new version of the *1186 User's Guide*. While many old problems have been resolved, some still remain. For that reason, a comment form is appended to each manual.

Chapters and sections in the two user's guides are now nearly parallel, and much of the content wherever appropriate, is identical, or nearly identical. Snaps of the various Execs and the background menu are included in Chapter 1.

The content of the old appendices is now distributed in the body of the text, and new appendices catalogue Library Modules and Fonts according to floppy disk location.

The chapter on diagnostics (Chapter 8) is largely new, and the local rigid disk file system has been broken out into a new chapter (Chapter 4). A new section on fonts has been added to the software installation procedures (Chapter 5). MP Codes are now listed separately as Chapter 9. Also, the cabling diagrams and other information related to RS-232 support formerly included in Chapter 7, Input/Output, is now in the *Lisp Library Modules Manual*.

Finally, in every chapter that requires use of Lisp expressions of any kind, there is a notice regarding the use of IL: and a suggestion that expressions, functions, and variables be typed into an Interlisp Exec.

1186 User's Guide Release Notes

What to Look For

The *1186 User's Guide* has been extensively reorganized and rewritten for the Lyric Release. Wherever possible, it is now nearly identical to the new version of the *1108 User's Guide*. While many old problems have been resolved, some still remain. For that reason, a comment form is appended to each manual.

Chapters and sections in the two user's guides are now nearly parallel, and much of the content wherever appropriate, is identical, or nearly identical. Snaps of the various Execs and the background menu are included in Chapter 1.

The content of the old appendices is now distributed in the body of the text, and new appendices catalogue Library Modules and Fonts according to floppy disk location.

The chapter on diagnostics (Chapter 8) are largely new, and the local rigid disk file system has been broken out into a new chapter (Chapter 4). A new section on fonts has been added to the software installation procedures (Chapter 5). Cursor Codes are now listed separately as Chapter 9.

Finally, in every chapter that requires use of Lisp expressions of any kind, there is a notice regarding the use of **IL:** and a suggestion that expressions, functions, and variables be typed into an Interlisp Exec.

7. KNOWN PROBLEMS

This chapter is a compilation of known problems in the Lyric release. These problems are in the form of Action Requests (ARs) from the Xerox Lisp AR data base. The appendix is organized by major Xerox Lisp categories: Communications, Windows and Graphics, Operating System, Language Support, Programming Environment, Common Lisp, System Tools, and Library. A brief description of the problem follows each AR number. Some ARs have specific workarounds, noted by an arrow (→).

Communications

Pup File Service

0012 Leaf sometimes prints "not responding" for non-user-visible operations, such as closing a cached file in the background.

0432 Leaf after logout activity is not interlocked against other processes; processes actively reading or writing a file at the time of LOGOUT may have problems.

3374 UNIX OPENFILE for access of APPEND or BOTH fails unless RECOG = OLD (this is a bug in the Leaf server).

5225 Read-Open Pup FTP files are not reopened after logout. The system prints "File has disappeared!" (even though it's only the server connection that has disappeared) and closes the stream.

XNS File Service

4453 DIRECTORY does not match wildcards against subdirectory names unless you include them in subdirectory syntax. For example, `<Fred>*S*` finds `<Fred>Doc>Case` but not `<Fred>Lisp>Hax`. `<Fred>*S*>` would find the latter but not the former.

4463 Confusion if you try to create an NS subdirectory with same name as an existing ordinary file—a given filename can designate a real file or a subdirectory, but not both. The errors signaled when trying to use a subdirectory as a file or vice versa are not always obvious.

7401 NS Filing says "Login incorrect" even if the error was "Cannot Authenticate". That is, your password may have been correctly specified, but the file service was unable to contact an authentication service within some timeout period.

7761 A file residing on an NS file server can be deleted despite there being an open input stream on it. The operations remain consistent, however—the input stream is read correctly to the end of file, but the file may no longer exist on the server when it finishes.

Other

7528 Changing NETWORKOSTYPES has no effect for host whose OS type has already been (incorrectly) defaulted. → Set it in your Init file on disk.

Windows and Graphics

Fonts & Hardcopy

0831 FILE NOT FOUND error in FONTCREATE not continuable. → After adding directories to whichever of DISPLAYFONDIRECTORIES or INTERPRESSFONTDIRECTORIES is appropriate, revert to FONTCREATE before saying OK.

3792 Interpress and Press BLTSHADE do not align textures (seams can occur).

4219 Interpress treats margins and clipping region of stream as synonymous.

4746 DSPFONT gives "Illegal argument" error instead of "Font not found" error when given a non-existent font name.

5618 Documentation: FONTCREATE ignores the *CHARSET* argument. (It is not needed, as non-zero character sets are created automatically on demand.)

5703 Underscore (ASCII #o137) prints as leftward arrow, which was the rendering of that character code in old ASCII, and still much used in Interlisp. → To make it print as underscore (in Interpress fonts), `(SETA \ASCIITONS 95 95)`. To restore previous behavior, `(SETA \ASCIITONS 95 172)`.

6322 HARDCOPYW with a rotation of 180 degrees places the image at the wrong coordinates.

Graphics

4771 DRAWCIRCLE with stream's DSPOPERATION set to INVERT fails if brush is bigger than 1.

4878 Bug in DRAWLINE: with DASHING, INVERT mode is ignored.

4879 Bug in DRAWLINE: with DASHING, the line width is larger than anticipated.

5647 SHADEITEM with shade WHITESHADE does not work for pop-up menus.

5721 If *SOURCE* argument to BITBLT is a display stream, *SOURCELEFT* and *SOURCEBOTTOM* arguments do not default to 0.

6502 DRAWELLIPSE goes into an infinite loop for some arguments. → Approximate it with DRAWCURVE.

6798 DRAWDASHEDLINE unnecessarily calls RELMOVETO.

Menus & Windows

4051 SUBITEM option of a menu doesn't work for multi-column menus.

6255 Submenus do not allow return to parent menu unless you come out to the left of them.

8629 Inspector prints at infinite depth when using the Interlisp read table. This can cause infinite loop or stack overflow if a structure is circular (special problem is stack frame backtraces). → (CHANGENAME 'PRINTANDBOX 'PRIN2 'CL:PRIN1)

Free Menu

6410 In Table or Column formatting, regions for special item highlighting aren't set properly.

7388 In some cases, FM.CHANGELABEL doesn't clear an item properly.

Operating System

File System

0340 OUTFILEP or FULLNAME/NEW with explicit version always returns NIL on {CORE}.

3817 SETFILEINFO of EOL attribute on CORE or disk stream sets only the stream's EOL, not the permanent file's EOL attribute. GETFILEINFO always fetches the permanent EOL attribute. → Set EOL for new files in call to OPENSTREAM; fix EOL for old files by SETFILEINFO on the file, not a stream on the file.

Floppy

3770 Break "RECORDNOTFOUND" occurs when floppy misformatted.

4042 Floppy in CPM mode has problems with EOL convention and CONTROL-Z as end of file.

4891 Floppy occasionally takes a long time to determine that it can't be formatted.

4992 FLOPPY.TO.FILE fails to specify type BINARY, so the image file may turn out TEXT. → (ADVISE '(OPENSTREAM :IN FLOPPY.TO.FILE) 'BEFORE '(PUSH OBSOLETE '(TYPE BINARY)))

5088 A break occurs if you try to copy to floppy a file that was created when the time was not set. → (SETFILEINFO *file* 'CREATIONDATE "*reasonable date*")

5212 1186 Floppy does not automatically notice new floppies, uses old cached information. → After changing floppies, DIR {FLOPPY} or CONN {FLOPPY} will update the cached information.

5259 1186 (FLOPPY.WAIT.FOR.FLOPPY) doesn't clear typeahead before user response.

7132 Floppy directory search with multiple wildcards is too slow.

7788 1108 Floppy directory cache is retained if machine is turned off in IDLE.

Local Disk

5014 Disk system on 29MB 1108 runs without error checking.

6434 Local file system does not accept the character "←" in file names.

Keyboard

6618 KEYACTION causes invalid address (MP 9305) if given an invalid *TABLE* argument.

Processes

7724 ADD.PROCESS creates unnecessary symbol for process name.

7886 PROCESS.EVAL should check for the process having been destroyed—if the process in which the evaluation occurs aborts or dies, caller hangs waiting for result. → Include an UNWIND-PROTECT in the evaluated form to do something interesting if an error occurs.

Other

0563 Time functions do not factor out disk swapout time accurately.

5568 Booting a virtual memory image saved by SAVEVM underneath Idle runs all the "after SAVEVM" tasks while still under Idle, which can cause problems for some devices.

7498 Lyric sysout invalidates VMem at startup time much faster than Koto did—you can't always immediately boot back to the original VMem state.

Language Support

Streams & I/O

3569 Printout commands .SUP, .SUB, .BASE do not work.

3889 PEEKC followed by READC prints an additional CR in the case where CR is input.

8186 LINELENGTH of synonym and other indirect streams (e.g., *QUERY-IO* and *ERROR-OUTPUT*) is not the same as the LINELENGTH of the underlying stream.

Storage Allocation & Garbage Collector

4935 Garbage collector can turn itself off when table fills because of many pointers getting reference count zero before a RECLAIM occurs (see the discussion in "Section 22.1, Storage Allocation and Garbage Collection" of the Release Notes, Chapter Changes to Interlisp-D).

5329 Reference count operations are slow on objects with large reference counts.

7008 STORAGE should print the type name using PRIN2.

Other

4348 (STRPOS "" "") returns 1 instead of NIL .

4349 RPLSTRING complains about Invalid argument for the replacing string, when it is really the index that is invalid.

6511 Soft stack overflow error can occur when there is still adequate stack space.

6955 DECLARERECORD goes into infinite loop if given a field specification of the form (*fieldname* 0 WORD).

Programming Environment

File Manager

2992 Asking FILES? to add a new filevar to filecoms adds the *contents* of the filevar, instead of creating the filevar.

4130 RENAME doesn't find occurrences of the old name in functions that have earlier been renamed, unless you do a MAKEFILE first.

4991 When MAKEFILE loads (DECLARE: DONTCOPY --) expressions from a previously non-loaded file, they all get evaluated regardless of their tags.

5878 Editing a variable whose VARTYPE is ALISTS now marks all entries as changed.

6606 Records redeclared under (LOAD & 'PROP) are not marked as changed, and no redeclaration warning is given.

7809 DELDEF of a Definer does not remove its prototype function.

7895 MAKEFILE accepts only symbols, not strings and pathnames, for filenames.

8227 Loading a FASL file does not add the file name to SYSFILES.

Editor

6563 SEdit's pretty-printer for special forms needs to be smarter about the possibility of backquoted forms appearing in unexpected places. E.g., `(let ,bindings ...) prettyprints poorly.

7745 SEdit doesn't grab the TTY when evalling an expression, so DWIM interactions, etc., require that user click mouse in tty window.

7928 Copy selecting a symbol you were just editing sometimes displays it in upper-case, independent of *print-case*.

7948 Can't copy select text of the form (QUOTE form) into SEdit—the parentheses matching is confused after SEdit turns it into 'form.

8018 If you attempt to copy select an expression into a form (cl:function) or (quote), a break occurs.

8257 ED <pathname> tries to edit the pathname structure instead of invoking the text editor on the file named by pathname.

8279 If INITIALS is NIL, edit time stamps are not created or replaced.

8378 Using Meta-X on nonexistent functions doesn't give any feedback.

8406 Meta-E command key should give better error message when applied to a left-button selection.

8480 SEdit pretty printer produces awkward indentation of CL:DO exit clauses.

Debugger

6981 Trace output doesn't stop when window is full.

7402 Closing a debugger window does not abort unless the window has the tty.

7445 InspectCode from the debugger backtrace window is only implemented for frames named by symbols with compiled code definitions. → To inspect the code behind a frame named si::*unwind-protect*, first evaluate (il:movd 'il:help 'si::*unwind-protect*).

7639 The error messages for Undefined Function and Unbound Variable print a trailing period after the function or variable name.

8139 Debugger's EDIT command does not reset context so that OK reevaluates from the right place.

Exec & TTYIN

7397 Since XCL:SET-EXEC-TYPE immediately sets *readtable*, *package*, etc., it should *not* be called from your Init file, as it will not have the intended effect; in fact, it will change the reading environment in which the Init file is being loaded. → Push onto the list POSTGREETFORMS an element of the form (XCL:SET-EXEC-TYPE "*type*").

7595 The implicit CL:PROCLAIM in CL:DEFVAR, CL:DEFPARAMETER, etc., is not undone by UNDO.

7751 ?= does not handle package errors.

7904 The NAME command misinterprets complex Event-Specs.

8383 ?= loops forever if there is a semicolon or unbalanced stringquote on the line. → Type CONTROL-E to get out.

8385 CONTROL-D or calling HARDRESET may change the Interlisp Exec's prompt.

8414 UNDO of multiple events gives misleading indication of what was undone—it only names one of the events.

8524 Exec Commands aren't passed all arguments when the first one is a list, since the Exec misinterprets the input as being in APPLY format.

Common Lisp

7260 The compiler does not permit binding more than 15 specials in a single LET or PROG.

8230 Optimizer for TYPEP does not catch errors in user-defined DEFTYPE expansion functions.

8340 The proceed case for EXPORT conflict errors doesn't actually export the symbol(s).

8622 Common Lisp interpreter does not support IL:OPENLAMBDA macros.

System Tools

4106 System Tools' mouse-confirm message is gibberish—should be "Click Left when ready, Right to exit".

5095 1186 System Tools: Keyboard on-line diagnostic shows wrong transitions on ESC key and middle mouse button.

6000 Sysin! in System Tools causes 915 if VmemSize not set.

6633 The Floppy Duplicate! command generates confusing error message: "Channel Status:goodCompletion Floppy disk error(write)". Ignore the bogus "goodCompletion".

7116 System Tools crashes with 915/935 whenever you try to access a volume that needs scavenging.

7420 Floppy Utility: must do Floppy Info! before List!

7446 Sysin to an 1186 from a VAX is twice as slow as to an 1108.

8386 The COPY VMEM! window is too small when using a 15" display.

8389 HELP command text scrolls up past top of message window on 15" display.

8403 MP 915 on Sysin if Organization set to 21 characters or more. The Clearinghouse specification requires that Domain and Organization names be 20 characters or less.

Library

4045

8153 4045XLPStream does not support international versions of the 4045 .

Chat

3360 Reshaping Chat window sometimes confuses terminal emulator—it starts typing halfway down window, bottom single line redisplayed repeatedly.

6586 VT100KP does not reinitialize CHATMENU with the new menu items. → Set CHATMENU to NIL after loading VT100KP.

8255 Screen mode reverse doesn't work in VTCHAT.

8325 Missing error types in GAP Courier program used by NSChat (if such errors occur, the error message contains an error number, rather than a description).

CopyFiles

8324 COPYFILES generates bogus directory when moving single file to explicitly named directoryless {CORE} or {FLOPPY} file. E.g., (COPYFILES "{FS}<LISP>TEST" "{CORE}MYTEST") writes the file {CORE}<MYTEST>TEST instead.

FileBrowser

5763 FileBrowser Rename to a numeric extension prints the renamed file as if the extension were the version.

7212 TableBrowser right button selection behavior is odd when selecting in the middle of non-contiguous selected files.

8061 FileBrowser's FB command requires that its keywords be in the Interlisp package.

FTPServer

6520 DIRECTORY cannot get highest version of a file from a machine running FTPSERVER.

6521 Cannot obtain the attributes CREATIONDATE, READDATE, WRITEDATE, LENGTH or PAGES from a machine running FTPSERVER.

8308 Writing a file to a machine running FTPSERVER with insufficient space for the file (on disk or floppy) can result in a loop with no error indication given.

FX80

8050 FASTFX80 prints 1 inch wider than FASTFX80.INCHES-PER-LINE specifies.

8317 HQFX80 printing does not reset line spacing to normal when it is finished.

Grapher

4336 GRAPHER doesn't call COPYFN on image object labels.

4778 LAYOUTSEXPR breaks if given non-NIL, non-list *BOXING* argument.

5383 EDITGRAPH node selection picks the node with the closest center point, not necessarily the node the cursor is actually over.

Kermit & Modem

7817 Kermit and Modem do not recover from timeouts or bad packets and eventually must abort.

KeyboardEditor

4725 Keyboard Editor can EDITCONFIGURATION a nonexistent configuration.

Masterscope

5614 Analyzing a file opens/closes the file once per function on the file, as it calls LOADFNS on each function. → Load the file PROP in the first place.

NSMaintain

6962 Change Password does not work. → Chat to a Clearinghouse server and use its Change Password command.

8165 NSMaintain incorrectly reports "done" even though operation not successfully performed.

8410 NSMaintain's Show Domain command supplies inappropriate default value (last user/group, not a domain).

RS232

6599 HARDRESET wedges TTY port.

Sketch

6899 Sketch Put and Get fail to close their files.

Spy

8138 Passing a mergetype argument of DEFAULT to SPY.TREE causes a break.

TCP

4303 DIR to a VMS host using TCP/IP lists no files.

5695 CONN to a Unix directory loses the last level of subdirectory.

6548 TCP can't be used to store files to a Unix directory that doesn't already exist. You must explicitly create the directory on the server side first.

8347 UNIX entry of NETWORKLOGINFO may need to use LF instead of CR for some hosts.

TEdit

4220 TEDIT.FIND searches beyond the given range; on large texts this results in very slow performance. → Use STRPOS on the result of TEDIT.SEL.AS.STRING instead.

4724 TEdit performance bad on big documents with many pieces.

4998 Big image objects don't display in the document after being selected into the document. → Reshape the window so the bitmap fits.

5100 READCCODE fails (non-numeric argument) when it encounters an image object on a TEdit stream.

5412 You can't print NS characters to a TEdit stream (you get an end of file error).

5619 You can't use the Find wildcards * and # in Substitute.

5707 Using the AGAIN key after the CASE key does not perform Case on the new selection, but replaces new selection with previous selection.

5729 Holding the shift key, then buttoning on a sketch, then in TEdit causes a break.

5758 TEdit middle button LOOKS command menus don't come up near the mouse.

5843 OPENTEXTSTREAM specifying START but with END = NIL gets zero characters.

5930 TEdit Selection display doesn't turn on when expected.

5993 TEdit prompt window does not push attached menus up when it expands.

6223 The LOOKS property in call to TEdit only sets the default font—not saved by Put. → Explicitly set the Looks after starting the edit.

6763 TEdit lost text due to timing problem during a delete selection.

7160 Abbreviation expansion doesn't get Looks right.

7243	TEdit CharLooks subwindow does not account for window title's height, so Character Looks window must now be scrolled to see its last line.
7499	Want TEdit's Undo command not to throw away text.
8294	PageLayout menu Show command returns message, "Format too complex to edit" for document formatted with TEDIT.PAGEFORMAT or PAGEFORMAT property.

TExec

6815	TExec doesn't handle NS chars well.
5771	TExec "Get" command should be removed.

VirtualKeyboards

4957	1186 "'" and "'" are not placed correctly for European keyboard.
4997	If you go into Idle mode while a VirtualKeyboard lacking English characters is in effect, you can't log back in.
6809	Standard-Russian Keyboard has wrong characters where capital CH and capital B should be.

[This page intentionally left blank]

APPENDIX A. THE EXEC

In most Common Lisp implementations, there is a "top-level **read-eval-print** loop," which reads an expression, evaluates it, and prints the results. In Xerox Common Lisp, the Exec acts as the top-level loop, but in addition to **read-eval-print**, it also performs a number of other tasks, and allows a much greater range of inputs.

The Exec is based on concepts from the Interlisp Programmer's Assistant (see the *Interlisp-D Reference Manual*).

The Exec traps all throws, and recovers gracefully. It prints all values resulting from evaluation, on separate lines. When zero values are returned, nothing is printed.

The Exec keeps track of your previous input, in a structure called the history list. A history list is a list of the information associated with each of the individual events that have occurred, where each event corresponds to one input. Associated with each event on the history list is the input, its values, plus other optional information such as side-effects, formatting information, etc.

The following dialogue contains illustrative examples and gives the flavor of the use of the Exec. Be sure to type these examples to an Exec whose ***PACKAGE*** is set to the **XCL-USER** package. The Exec that Lisp starts up with is set to the **XCL-USER** package. Each prompt consists of an event number and a prompt character (">").

```
12>(setq foo 5)
5
13>(setq foo 10)
10

14>undo[cr]
SETQ undone.
15>foo[cr]
5
```

This is an example of direct communication with the Exec. You have instructed the Exec to **undo** *the previous event.*

```
...
25>set(lst1 (a b c))
(A B C)
26>(setq lst2 '(d e f))
(D E F)
27>(mapc #'(lambda (x) (setf (get x 'myprop) t)) lst1)
(A B C)
```

The Exec accepts input both in APPLY format (the **SET***) and EVAL format (the* **SETQ***.) In event 27, the user adds a property MYPROP to the symbols* **A**, **B**, *and* **C**.

```
28>use lst2 for lst1 in 27[cr]
NIL
```

You just instructed the Exec to go back to event number 27, substitute **LST2** *for* **LST1**, *and then re-execute the expression. You could have also used -2 instead of 27, specifying a relative address.*

.
.
.

46>**(setf my-hash-table (make-hash-table))**

```
#<Hash-Table @ 66,114034>
```

47>**(setf (gethash 'foo my-hash-table) (string 'foo))**

```
"FOO"
```

If **STRING** *were computationally expensive (which it is not), then you might be caching its value for later use.*

48>**use fie for foo in string**cr

```
"FIE"
```

You now decide you would like to redo the **SETF** *with a different value. You specify the event using* **"IN STRING"** *rather than* **SETF**.

49>**?? use**cr

```
   USE FIE FOR FOO IN STRING
48>  (SETF (GETHASH 'FIE MY-HASH-TABLE)
       (STRING 'FIE))
       "FIE"
```

Here you ask the Exec (using the ?? command) what it has on its history list for the last input. Since the event corresponds to a command, the Exec displays both the original command and the generated input.

The most common interaction with the Exec occurs at the top level or in the debugger, where you type in expressions for evaluation, and see the values printed out. In this mode, the Exec acts much like a standard Common Lisp top-level loop, except that before attempting to evaluate an input, the Exec first stores it in a new entry on the history list. Thus if the operation is aborted or causes an error, the input is still saved and available for modification and/or re-execution. The Exec also notes new functions and variables to be added to its spelling lists to enable future corrections.

After updating the history list, the Exec executes the computation (i.e., evaluates the form or applies the function to its arguments), saves the value in the entry on the history list corresponding to the input, and prints the result. Finally the Exec displays a prompt to indicate it is again ready for input.

Input Formats

The Exec accepts three forms of input: an expression to be evaluated (EVAL-format), a function-name and arguments to apply it to (APPLY-format), and Exec commands, as follows:

EVAL-format input

If you type a single expression, either followed by a carriage-return, or, in the case of a list, terminated with balanced parenthesis, the expression is evaluated and the value is returned. For example, if the value of the variable **FOO** is the list **(A B C)**:

```
32>FOOcr
(A B C)
```

Similarly, if you type a Lisp expression, beginning with a left parenthesis and terminated by a matching right parenthesis, the form is simply passed to **EVAL** for evaluation. Notice that it is not necessary to type a carriage return at the end of such a form; the reader will supply one automatically. If a carriage-return is typed before the final matching right parenthesis or bracket, it is treated the same as a space, and input continues. The following examples are interpreted identically:

```
123> (+ 1 (* 2 3))
7
124> (+ 1 (*cr
2 3))
7
```

APPLY-format input

Often, when typing at the keyboard, you call functions with constant argument values, which would have to be quoted if you typed them in "EVAL-format." For convenience, if you type a symbol immediately followed by a list form, the symbol is **APPLY**ed to the elements within the list, unevaluated. The input is terminated by the matching right parenthesis. For example, typing **LOAD(FOO)** is equivalent to typing **(LOAD 'FOO)**, and **GET(X COLOR)** is equivalent to **(GET 'X 'COLOR)**. As a simple special case, a single right parenthesis is treated as a balanced set of parentheses, e.g.

```
125>UNBREAK)
```

is equivalent to

```
125>UNBREAK()
```

The reader will only supply the "carriage return" automatically if no space appears between the initial symbol and the list that follows; if there is a space after the initial symbol on the line and the list that follows, the input is not terminated until a carriage return is explicitly typed.

Note that APPLY-format input cannot be used for macros or special forms.

Exec commands

The Exec recognizes a number of commands, which usually refer to past events on the history list. These commands are treated specially; for example, they may not be put on the history list. The format of a command is always a line beginning with the command name. (The Exec looks up the command name independent of package, so that Exec commands are package independent.) The remainder of the line, if any, is treated as "arguments" to the command. For example,

```
128>UNDOcr
mapc undone
129>UNDO (FOO --)cr
foo undone
```

are all valid command inputs.

Multiple Execs and the Exec's Type

Multiple Execs

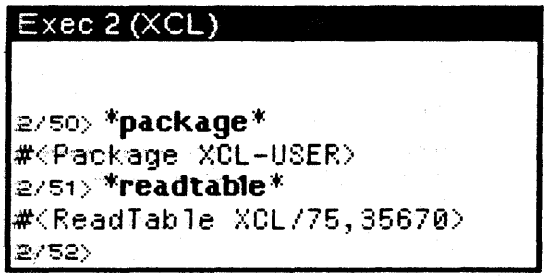
More than one Exec can be active at any one time. New Execs can be created by selecting the Exec menu item in the background pop-up menu. When a prompt is printed for an event in other than the first Exec, the prompt is preceded with the Exec number; for example:

```
2/50>
```

might be a prompt in Exec 2. All Execs share the same history list, but each event records which Exec it goes with. That is, although a single global list exists, the Xerox Lisp history system maintains the separate threads of control within each Exec.

Exec type

Several variables are very important to an Exec since they control the format of reading and printing. Together these variables describe a type of exec. Put another way, this is the Exec's mode. To allow easier setting of these modes some standard bindings for the variables have been named. The names provide the user an Exec of the Common Lisp (CL), Interlisp (IL) or Xerox Extended Common Lisp (XCL) type. An Exec's type is usually displayed in the title bar of its window in parentheses:



Event Specification

Exec commands, like **UNDO**, frequently refer to previous events in the session's history. All Exec commands use the same conventions and syntax for indicating which event(s) the command refers to. This section shows you the syntax used to specify previous events.

An event address identifies one event on the history list. For example, the event address **42** refers to the event with event number 42, and **-2** refers to two events back in the *current* Exec. Usually, an event address will contain only one or two commands.

Event addresses can be concatenated. For example, if **FOO** refers to event ***N***, **FOO FIE** will refer to the first event before event ***N*** which contains **FIE**.

The symbols used in event addresses (such as **AND**, **F**, **=**, etc. are compared with **STRING-EQUAL**, so that it does not matter what the current package is when you type an event address symbol to an Exec.

Event addresses are interpreted as follows:

N (an integer)

If ***N*** is positive, it refers to the event with event number ***N*** (no matter which Exec the event occurred in.) If ***N*** is negative, it

always refers to the event *-N* events backwards counting *only* events belonging to the *current* Exec.

F Specifies that the next object in the event address is to be searched for, regardless of what it is. For example, **F -2** looks for an event containing **-2**.

= Specifies that the next object is to be searched for in the *values* of events, instead of the inputs.

SUCHTHAT *PRED* Specifies an event for which the function *PRED* returns true. *PRED* should be a function of two arguments, the input portion of the event, and the event itself.

PAT Any other event address command specifies an event whose input contains an expression that matches *PAT*. When multiple Execs are active, all events are searched, no matter which Exec they belong to. The pattern can be a simple symbol, or a more complex search pattern.

Note: Specifications used below of the form $EventAddress_i$ refer to event addresses, as described above. Since an event address may contain multiple words, the event address is parsed by searching for the words which delimit it. For example, in $EventAddress_1$ **AND** $EventAddress_2$, the notation $EventAddress_1$ corresponds to all words up to the **AND** in the event specification, and $EventAddress_2$ to all words after the **AND** in the event specification.

FROM *EventAddress* All events since *EventAddress*, inclusive. For example, if there is a single Exec and the current event is number 53, then **FROM 49** specifies events 49, 50, 51, and 52. **FROM** will include events from *all* Execs.

ALL *EventAddress* Specifies all events satisfying *EventAddress*. For example, **ALL LOAD, ALL SUCHTHAT FOO-P**.

empty If nothing is specified, it is the same as specifying **-1**, i.e., the last event in the current Exec.

$EventSpec_1$ **AND** $EventSpec_2$ **AND . . . AND** $EventSpec_N$

Each of the $EventSpec_i$ is an event specification. The lists of events are concatenated. For example, **ALL MAPC AND ALL STRING AND 32** specifies all events containing **MAPC**, all containing **STRING**, and also event **32**. Duplicate events are removed.

Exec Commands

All Exec commands are input as lines which begin with the name of the command. The name of an Exec command is not a symbol and therefore is not sensitive to the setting of the current package (the value of ***PACKAGE***).

EventSpec is used to denote an event specification which in most cases will be either a specific event address (e.g., 42) or a relative one (e.g., -3). Unless specified otherwise, omitting *EventSpec* is

the same as specifying *EventSpec* = **-1**. For example, **REDO** and **REDO -1** are the same.

REDO *EventSpec* [Exec command]

Redoes the event or events specified by *EventSpec*. For example, **REDO 123** redoes the event numbered 123.

RETRY *EventSpec* [Exec command]

Similar to **REDO** except sets the debugger parameters so that any errors that occur while executing *EventSpec* will cause breaks.

USE *NEW* **[FOR** *OLD* **] [IN** *EventSpec* **]** [Exec command]

Substitutes *NEW* for *OLD* in the events specified by *EventSpec*, and redoes the result. *NEW* and *OLD* can include lists or symbols, etc.

For example, **USE SIN (- X) FOR COS X IN -2 AND -1** will substitute **SIN** for every occurrence of **COS** in the previous two events, and substitute **(- X)** for every occurrence of **X**, and reexecute them. (The substitutions do not change the previous information saved about these events on the history list.)

If **IN** *EventSpec* is omitted, the first member of *OLD* is used to search for the appropriate event. For example, **USE DEFAULTFONT FOR DEFLATFONT** is equivalent to **USE DEFAULTFONT FOR DEFLATFONT IN F DEFLATFONT**. The **F** is inserted to handle correctly the case where the first member of *OLD* could be interpreted as an event address command.

If *OLD* is omitted, substitution is for the "operator" in that command. For example **FBOUNDP(FF)** followed by **USE CALLS** is equivalent to **USE CALLS FOR FBOUNDP IN -1**.

If *OLD* is not found, **USE** will print a question mark, several spaces and the pattern that was not found. For example, if you specified USE Y FOR X IN 104 and X was not found, "X ?" is printed to the Exec.

You can also specify more than one substitution simultaneously as follows:

USE NEW_1 **FOR** OLD_1 **AND ... AND** NEW_N **FOR** OLD_N **[IN** *EventSpec* [Exec command]

Note: The **USE** command is parsed by a small finite state parser to distinguish the expressions and arguments. For example, **USE FOR FOR AND AND AND FOR FOR** will be parsed correctly.

Every **USE** command involves three pieces of information: the expressions to be substituted, the arguments to be substituted for, and an event specification that defines the input expression in which the substitution takes place. If the **USE** command has the same number of expressions as arguments, the substitution procedure is straightforward. For example, **USE X Y FOR U V**

means substitute **X** for **U** and **Y** for **V**, and is equivalent to **USE X FOR U AND Y FOR V**.

However, the **USE** command also permits distributive substitutions for substituting several expressions for the same argument. For example, **USE A B C FOR X** means first substitute **A** for **X** then substitute **B** for **X** (in a new copy of the expression), then substitute **C** for **X**. The effect is the same as three separate **USE** commands.

Similarly, **USE A B C FOR D AND X Y Z FOR W** is equivalent to **USE A FOR D AND X FOR W**, followed by **USE B FOR D AND Y FOR W**, followed by **USE C FOR D AND Z FOR W**. **USE A B C FOR D AND X FOR Y** also corresponds to three substitutions, the first with **A** for **D** and **X** for **Y**, the second with **B** for **D**, and **X** for **Y**, and the third with **C** for **D**, and again **X** for **Y**. However, **USE A B C FOR D AND X Y FOR Z** is ambiguous and will cause an error.

Essentially, the **USE** command operates by proceeding from left to right handling each **AND** separately. Whenever the number of expressions exceeds the number of expressions available, multiple **USE** expressions are generated. Thus **USE A B C D FOR E F** means substitute **A** for **E** at the same time as substituting **B** for **F**, then in another copy of the indicated expression, substitute **C** for **E** and **D** for **F**. This is also equivalent to **USE A C FOR E AND B D FOR F**.

Note: The **USE** command correctly handles the situation where one of the old expressions is the same as one of the new ones, **USE X Y FOR Y X**, or **USE X FOR Y AND Y FOR X**.

? &OPTIONAL *NAME* [Exec command]

If *NAME* is not provided describes all available Exec commands by printing the name, argument list, and description of each. With *NAME*, only that command is described.

?? *EventSpec* [Exec command]

Prints the most recent event matching the given *EventSpec*.

CONN *DIRECTORY* [Exec command]

Changes default pathname to *DIRECTORY*.

DA [Exec command]

Returns current date and time.

DIR &OPTIONAL *PATHNAME* &REST *KEYWORDS* [Exec command]

Shows a directory listing for *PATHNAME* or the connected directory. If provided, *KEYWORDS* indicate information to be displayed for each file. Some keywords are: AUTHOR, AU, CREATIONDATE, DA, etc.

DO-EVENTS &REST *INPUTS* &ENVIRONMENT *ENV* [Exec command]

DO-EVENTS is intended as a way of putting together several different events, which can include commands. It executes the multiple *INPUTS* as a single event. The values returned by the **DO-EVENTS** event are the concatenation of the values of the inputs. An input is not an EventSpec, but a call to a function or command. If *ENV* is provided it is a lexical environment in which all evaluations (functions and commands) will take place. Event specification in the *INPUTS* should be explicit, not relative, since referring to the last event will reinvoke the executing **DO-EVENTS** command.

FIX &REST *EventSpec* [Exec command]

Edits the specified event prior to reexecuting it. If the number of characters in the Fixed line is less than the variable **TTYINFIXLIMIT** then it will be edited using TTYIN, otherwise the Lisp editor is called via **EDITE**.

FORGET &REST *EventSpec* [Exec command]

Erases **UNDO** information for the specified events.

NAME *COMMAND-NAME* &OPTIONAL *ARGUMENTS* &REST *EVENT-SPEC* [Exec command]

Defines a new command, *COMMAND-NAME*, and its *ARGUMENTS*, containing the events in *EVENT-SPEC*.

NDIR &OPTIONAL *PATHNAME* &REST *KEYWORDS* [Exec command]

Shows a directory listing for *PATHNAME* or the connected directory in abbreviated format. If provided, *KEYWORDS* indicate information to be displayed for each file. Some keywords are: AUTHOR, AU, CREATIONDATE, DA, etc.

PL *SYMBOL* [Exec command]

Prints the property list of *SYMBOL* in an easy to read format.

REMEMBER &REST *EVENT-SPEC* [Exec command]

Tells File Manager to remember type-in from specified event(s) , *EVENT-SPEC* , as expressions to save.

SHH &REST *LINE* [Exec command]

Executes *LINE* without history list processing.

UNDO &REST *EventSpec* [Exec command]

Undoes the side effects of the specified event (see below under "Undoing").

PP &OPTIONAL *NAME* &REST *TYPES* [Exec command]

Shows (prettyprinted) the definitions for *NAME* specified by *TYPES*.

SEE &REST *FILES* [Exec command]

Prints the contents of *FILES* in the Exec window, hiding comments.

SEE* &REST *FILES* [Exec command]

Prints the contents of *FILES* in the Exec window, showing comments.

TIME *FORM* &KEY *REPEAT* &ENVIRONMENT *ENV* [Exec command]

Times the evaluation of *FORM* in the lexical environment *ENV*, repeating *REPEAT* number of times. Information is displayed in the Exec window.

TY &REST *FILES* [Exec command]

Exactly like the **TYPE** Exec command.

TYPE &REST *FILES* [Exec command]

Prints the contents of *FILES* in the Exec window, hiding comments.

Variables

A number of variables are provided for convenience in the Exec.

IL:IT [Variable]

Whenever an event is completed, the global value of the variable **IT** is reset to the event's value. For example,

```
312>(SQRT 2)
1.414214
313>(SQRT IL:IT)
1.189207
```

Following a **??** command, **IL:IT** is set to the value of the last event printed. The inspector has an option for setting the variable **IL:IT** to the current selection or inspected object, as well. The variable **IL:IT** is global, and is shared among all Execs. **IL:IT** is a convenient mechanism for passing values from one process to another.

Note: **IT** is in the INTERLISP package and these examples are intended for an Exec whose ***PACKAGE*** is set to **XCL-USER**. Thus, **IT** must be package qualified (the **IL:**).

The following variables are maintained independently by each Exec. (When a new Exec is started, the initial values are **NIL**, or, for a nested Exec, the value for the "parent" Exec. However, events executed under a nested Exec will not affect the parent values.)

CL:- [Variable]

CL:+ [Variable]

CL:+ + [Variable]

CL:+ + + [Variable]

While a form is being evaluated by the Exec, the variable - is bound to the form, **CL:+** is bound to the previous form, **CL:+ +** the one before, etc. If the input is in apply-format rather than eval-format, the value of the respective variable is just the function name.

CL:* [Variable]

CL:** [Variable]

CL:*** [Variable]

While a form is being evaluated by the Exec, the variable **CL:*** is bound to the (first) value returned by the last event, **CL:**** to the event before that, etc. The variable **CL:*** differs from **IT** in that **IT** is global while each separate Exec maintains its own copy of **CL:***, **CL:**** and **CL:*****. In addition, the history commands change **IT**, but only inputs which are retained on the history list can change **CL:***.

CL:/ [Variable]

CL:// [Variable]

CL:/// [Variable]

While a form is being evaluated by an Exec, the variable **CL:/** is bound to a list of the results of the last event in that Exec, **CL://** to the values of the event before that, etc.

Fonts in the Exec

The Exec can use different fonts for displaying the prompt, user's input, intermediate printout, and the values returned by evaluation. The following variables control the Exec's font use:

PROMPTFONT [Variable]

Font used for printing the event prompt.

INPUTFONT [Variable]

Font used for echoing user's type-in.

PRINTOUTFONT [Variable]

Font used for any intermediate printing caused by execution of a command or evaluation of a form. Initially the same as **DEFAULTFONT**.

VALUEFONT [Variable]

Font used to print the values returned by evaluation of a form. Initially the same as **DEFAULTFONT**.

Changing the Exec

(CHANGESLICE *N HISTORY* **—)** [Function]

Changes the time-slice of the history list *HISTORY* to *N*. If **NIL**, *HISTORY* defaults to the top level history **LISPXHISTORY**.

Note: The effect of *increasing* the time-slice is gradual: the history list is simply allowed to grow to the corresponding length before any events are forgotten. *Decreasing* the time-slice will immediately remove a sufficient number of the older events to bring the history list down to the proper size. However, **CHANGESLICE** is undoable, so that these events are (temporarily) recoverable. Therefore, if you want to recover the storage associated with these events without waiting *N* more events until the **CHANGESLICE** event drops off the history list, you must perform a **FORGET** command.

Defining New Commands

You can define new Exec commands using the **XCL:DEFCOMMAND** macro.

(XCL:DEFCOMMAND *NAME ARGUMENT-LIST &REST BODY***)** [Macro]

XCL:DEFCOMMAND is similar to **XCL:DEFMACRO**, but defines new Exec commands. The *ARGUMENT-LIST* can have keywords, defstructure, and use all of the features of macro argument lists. When *NAME* is subsequently typed to the Exec, the rest of the line is processed like the arguments to a macro, and the *BODY* is executed. **XCL:DEFCOMMAND** is a definer; the File Manager will remember typed-in definitions and allow them to be saved, edited with **EDITDEF**, etc.

There are actually three kinds of commands that can be defined, **:EVAL**, **:QUIET**, and **:INPUT**. Commands can also be marked as only for the debugger, in which case they are labelled as **:DEBUGGER**. The command type is noted by supplying a list for the *NAME* argument to **XCL:DEFCOMMAND**, where the first element of the list is the command name, and the other elements are keyword(s) for the command type and, optionally **:DEBUGGER**.

Note: The documentation string in user defined Exec commands is automatically added to the documentation

descriptions by the **CL:DOCUMENTATION** function under the COMMANDS type and can be shown using the **?** Exec command.

:EVAL This is the default. The body of the command just gets executed, and its value is the value of the event. For example (in an XCL Exec),

```
(DEFCOMMAND (LS :EVAL)
(&OPTIONAL (NAMESTRING *DEFAULT-PATHNAME-DEFAULTS*)
&REST DIRECTORY-KEYWORDS)
(MAPC
        #'(LAMBDA (PATHNAME) (FORMAT T "~&~A" (NAMESTRING PATHNAME)))
        (APPLY #'DIRECTORY NAMESTRING DIRECTORY-KEYWORDS))
(VALUES))
```

would define the **LS** command to print out all file names that match the input namestring. The **(VALUES)** means that no value will be printed by the event, only the intermediate output from the **FORMAT**.

:QUIET These commands are evaluated, but neither your input nor the results of the command are stored on the history list. For example, the **??** and **SHH** commands are quiet.

:INPUT These commands work more like macros, in that the result of evaluating the command is treated as a new line of input. The **FIX** command is an input command. The result is treated as a line; a single expression in EVAL-format should be returned as a list of the expression to **EVAL**.

Undoing

Note: This discussion only applies to undoing under the Exec, Debugger and within the UNDOABLY macro; editors handle undoing in a different fashion.

The **UNDO** facility allows recording of destructive changes such that they can be played back to restore a previous state. There are two kinds of **UNDO**ing: one is done by the Exec, the other is available for use in a programmer's code. Both methods share information about what kind of operations can be undone and where the changes are recorded.

Undoing in the Exec

UNDO *EventSpec* [Exec command]

The Exec's **UNDO** command is implemented by watching the evaluation of forms and requiring undoable operations in that evaluation to save enough information on the history list to reverse their side effects. The Exec simply executes operations, and any undoable changes that occur are automatically saved on the history list by the responsible functions. The **UNDO** command works on itself the same way: it recovers the saved information and performs the corresponding inverses. Thus, **UNDO** is effective on itself, so that you can **UNDO** an **UNDO**, and **UNDO** that, etc.

Only when you attempt to undo an operation does the Exec check to see whether any information has been saved. If none has been saved, and you have specifically named the event you want undone, the Exec types **nothing saved**. (When you just type **UNDO**, the Exec only tries to undo the last operation.)

UNDO watches evaluation using **CL:EVALHOOK** (thus, calling **CL:EVALHOOK** cannot be undone). Each form given to **EVAL** is examined against the list **LISPXFNS** to see if it has a corresponding undoable version. If an undoable version of a call is found, it is called with the same arguments instead of the original. Therefore, before evaluating all subforms of your input, the Exec substitutes the corresponding undoable call for any destructive operation. For example, if you type **(DEFUN FOO ...)**, undoable versions of the forms that set the definition into the symbol function cell are evaluated. **FOO**'s function definition itself is not made undoable.

Undoing in Programs

There are two ways to make a program undoable. The simplest method is to wrap the program's form in the **UNDOABLY** macro. The other is to call undoable versions of destructive operations directly.

(XCL:UNDOABLY *&REST FORMS*) [Macro]

Executes the forms in *FORMS* using undoable versions of all destructive operations. This is done by "walking" (see **WALKFORM**) all of the *FORMS* and rewriting them to use the undoable versions of destructive operations (**LISPXFNS** makes the association).

(STOP-UNDOABLY *&REST FORMS* **)** [Macro]

Normally executes as **PROGN**; however, within an **UNDOABLY** form, explicitly causes *FORMS* not to be done undoably. Turns off rewriting of the *FORMS* to be undoable inside an **UNDOABLY** macro.

Undoable Versions of Common Functions

Efficiency and overhead are serious considerations for the execution of a user program. Thus, the programmer may need more control over the saving of undo information than that provided by the **UNDOABLY** macro.

To make a function undoable, you can simply substitute the corresponding undoable function if you want to make a destructive operation in your own program undoable. When the undoable function is called, it will save the undo information in the current event on the history list.

Various operations, most notably **SETF**, have undoable versions. The following undoable macros are initially available:

UNDOABLY-POP
UNDOABLY-PUSH
UNDOABLY-PUSHNEW
UNDOABLY-REMF
UNDOABLY-ROTATEF
UNDOABLY-SHIFTF
UNDOABLY-DECF
UNDOABLY-INCF
UNDOABLY-SET-SYMBOL
UNDOABLY-MAKUNBOUND
UNDOABLY-FMAKUNBOUND
UNDOABLY-SETQ
XCL:UNDOABLY-SETF
UNDOABLY-PSETF
UNDOABLY-SETF-SYMBOL-FUNCTION
UNDOABLY-SETF-MACRO-FUNCTION

Note: Many destructive Common Lisp functions do not currently have undoable versions, e.g., **CL:NREVERSE**, **CL:SORT**, etc. The current list of undoable functions is saved on the association list **LISPXFNS**.

Modifying the UNDO Facility

You will usually wish to extend the **UNDO** facility after creating a form whose side effects it might be desirable to undo, for instance a file renaming function.

An undoable version of the function needs to be written. This can be done by explicitly saving previous state information away, or by renaming calls in the function to their undoable equivalent. Undo information should be saved on the history list using **IL:UNDOSAVE**.

You must then hook the undoable version of the function into the undo facility. You do this by either using the **IL:LISPXFNS** association list, or in the case of a **SETF** modifier, on the **IL:UNDOABLE-SETF-INVERSE** property of the **SETF** function.

LISPXFNS [Variable]

Contains an association list which maps from destructive operations to their undoable form. Initially this list contains:

((CL:POP . UNDOABLY-POP)
(CL:PSETF . NDOABLY-PSETF)
(CL:PUSH . UNDOABLY-PUSH)
(CL:PUSHNEW . UNDOABLY-PUSHNEW)
((CL:REMF) . UNDOABLY-REMF)
(CL:ROTATEF . UNDOABLY-ROTATEF)
(CL:SHIFTF . UNDOABLY-SHIFTF)
(CL:DECF . UNDOABLY-DECF)

(CL:INCF . UNDOABLY-INCF)

(CL:SET . UNDOABLY-SET-SYMBOL)

(CL:MAKUNBOUND . UNDOABLY-MAKUNBOUND)

(CL:FMAKUNBOUND . UNDOABLY-FMAKUNBOUND)

. . . plus the original Interlisp undo associations)

(XCL:UNDOABLY-SETF *PLACE VALUE ...* **)** [Macro]

Like **CL:SETF** but saves information so it may be undone. **UNDOABLY-SETF** uses undoable versions of the setf function located on the **UNDOABLE-SETF-INVERSE** property of the function being **SETF**ed. Initially these **SETF** names have such a property:

CL:SYMBOL-FUNCTION - UNDOABLY-SETF-SYMBOL-FUNCTION

CL:MACRO-FUNCTION - UNDOABLY-SETF-MACRO-FUNCTION

(UNDOABLY-SETQ *&REST FORMS***)** [Function]

Typed-in **SETQ**s (and **SETF**s on symbols) are made undoable by substituting a call to **UNDOABLY-SETQ**. **UNDOABLY-SETQ** operates like **SETQ** on lexical variables or those with dynamic bindings; it only saves information on the history list for changes to global, "top-level" values.

(UNDOSAVE *UNDOFORM HISTENTRY***)** [Function]

Adds the undo information *UNDOFORM* to the **SIDE** property of the history event *HISTENTRY*. If there is no **SIDE** property, one is created. If the value of the **SIDE** property is **NOSAVE**, the information is not saved. *HISTENTRY* specifies an event. If *HISTENTRY* = **NIL**, the value of **LISPXHIST** is used. If both *HISTENTRY* and **LISPXHIST** are **NIL**, **UNDOSAVE** is a no-op.

The form of *UNDOFORM* is **(***FN . ARGS***)**. Undoing is done by performing **(APPLY (CAR** *UNDOFORM***) (CDR** *UNDOFORM***))**.

\#UNDOSAVES [Variable]

The value of **\#UNDOSAVES** is the maximum number of *UNDOFORM*s to be saved for a single event. When the count of *UNDOFORM*s reaches this number, **UNDOSAVE** prints the message **CONTINUE SAVING?**, asking if you want to continue saving. If you answer **NO** or default, **UNDOSAVE** discards the previously saved information for this event, and makes **NOSAVE** be the value of the property **SIDE**, which disables any further saving for this event. If you answer **YES**, **UNDOSAVE** changes the count to -1, which is then never incremented, and continues saving. The purpose of this feature is to avoid tying up large quantities of storage for operations that will never need to be undone.

If **\#UNDOSAVES** is negative, then when the count reaches **(ABS \#UNDOSAVES)**, **UNDOSAVE** simply stops saving without printing any messages or other interactions.

\#UNDOSAVES = NIL is equivalent to **\#UNDOSAVES** = infinity. **\#UNDOSAVES** is initially **NIL**.

The configuration described here has been found to be a very satisfactory one. You pay a very small price for the ability to undo what you type in, since the interpreted evaluation is simply watched for destructive operations, or if you wish to protect yourself from malfunctioning in your own programs, you can explicitly call, or have your program rewritten to explicitly call, undoable functions.

Undoing Out of Order

UNDOABLY-SETF operates undoably by saving (on the history list) the cell that is to be changed and its original contents. Undoing an **UNDOABLY-SETF** restores the saved contents.

This implementation can produce unexpected results when multiple modifications are made to the same piece of storage and then undone out of order. For example, if you type **(SETF (CAR FOO) 1)**, followed by **(SETF (CAR FOO) 2)**, then undo both events by undoing the most recent event first, then undoing the older event, **FOO** will be restored to its state before either event operated. However if you undo the first event, *then* the second event, **(CAR FOO)** will be **1**, since this is what was in **CAR** of **FOO** before **(UNDOABLY-SETF (CAR FOO) 2)** was executed. Similarly, if you type **(NCONC FOO '(1))**, followed by **(NCONC FOO '(2))**, undoing just **(NCONC FOO '(1))** will remove both **1** and **2** from **FOO**. The problem in both cases is that the two operations are not independent.

In general, operations are always independent if they affect different lists or different sublists of the same list. Undoing in reverse order of execution, or undoing independent operations, is always guaranteed to do the right thing. However, undoing dependent operations out of order may not always have the predicted effect.

Format and Use of the History List

LISPXHISTORY [Variable]

The Exec currently uses one primary history list, **LISPXHISTORY** for the storing events.

The history list is in the form (*EVENTS EVENT# SIZE MOD*), where *EVENTS* is a list of events with the most recent event first, *EVENT#* is the event number for the most recent event on *EVENTS*, *SIZE* is the the maximum length *EVENTS* is allowed to grow. *MOD* is is the maximum event number to use, after which event numbers roll over. **LISPXHISTORY** is initialized to **(NIL 0 100 1000)**.

The history list has a maximum length, called its time-slice. As new events occur, existing events are aged, and the oldest events are forgotten. The time-slice can be changed with the function **CHANGESLICE**. Larger time-slices enable longer memory spans, but tie up correspondingly greater amounts of storage. Since a

user seldom needs really ancient history, a relatively small time-slice such as 30 events is usually adequate, although some users prefer to set the time-slice as large as 200 events.

Each individual event on *EVENTS* is a list of the form (*INPUT ID VALUE . PROPS*). For Exec events, *ID* is a list (*EVENT-NUMBER EXEC-ID*). The *EVENT-NUMBER* is the number of the event, while the *EXEC-ID* is a string that uniquely identifies the Exec. (The *EXEC-ID* is used to identify which events belong to the "same" Exec.) *VALUE* is the (first) value of the event. *PROPS* is a property list used to associate other information with the event (described below).

INPUT is the input sequence for the event. Normally, this is just the input that the user typed-in. For an APPLY-format input this is a list consisting of two expressions; for an EVAL-format input, this is a list of just one expression; for an input entered as list of atoms, *INPUT* is simply that list. For example,

User Input	*INPUT* is:
LIST(1 2)	**(LIST (1 2))**
(LIST 1 1)	**((LIST 1 1))**
DIR "{DSK}<LISPFILES>"^{cr}	**(DIR "{DSK}<LISPFILES>")**

If you type in an Exec command that executes other events (**REDO**, **USE**, etc.), several events might result. When there is more than one input, they are wrapped together into one invocation of the **DO-EVENTS** command.

The same convention is used for representing multiple inputs when a **USE** command involves sequential substitutions. For example, if you type **FBOUNDP(FOO)** and then **USE FIE FUM FOR FOO**, the input sequence that will be constructed is **DO-EVENTS (EVENT FBOUNDP (FIE)) (EVENT FBOUNDP (FUM))**, which is the result of substituting **FIE** for **FOO** in **(FBOUNDP (FOO))** concatenated with the result of substituting **FUM** for **FOO** in **(FBOUNDP (FOO))**.

PROPS is a property list of the form (*PROPERTY*$_1$ *VALUE*$_1$ *PROPERTY*$_2$ *VALUE*$_2$ ...), that can be used to associate arbitrary information with a particular event. Currently, the following properties are used by the Exec:

SIDE A list of the side effects of the event. See **UNDOSAVE**.

LISPXPRINT Used to record calls to **EXEC-FORMAT**, and printed by the ?? command.

Making or Changing an Exec

(XCL:ADD-EXEC &KEY *PROFILE REGION TTY ID***)** [Function]

Creates a new process and window with an Exec running in it. *PROFILE* is the type of the Exec to be created (see below under XCL:SET-EXEC-TYPE). *REGION* optionally gives the shape and location of the window to be used. If not provided the user will be prompted. *TTY* is a flag, which, if true, causes the tty to be

given to the new Exec process. *ID* is a string identifier to use for events generated in this exec. *ID* defaults to the number given to the Exec process created.

(XCL:EXEC *&KEY WINDOW PROMPT COMMAND-TABLES ENVIRONMENT PROFILE TOP-LEVEL-P TITLE FUNCTION ID***)** [Function]

This is the main entry to the Exec. The arguments are:

WINDOW defaults to the current TTY display stream, or can be provided a window in which the Exec will run.

PROMPT is the prompt to print.

COMMAND-TABLES is a list of hash-tables for looking up commands (e.g., ***EXEC-COMMAND-TABLE*** or ***DEBUGGER-COMMAND-TABLE***).

ENVIRONMENT is a lexical environment used to evaluate things in.

READTABLE is the default readtable to use (defaults to the "Common Lisp" readtable).

PROFILE is a way to set the Exec's type (see above, "Multiple Execs and the Exec's Type").

TOP-LEVEL-P is a boolean, which should be true if this Exec is at the top level.

TITLE is an identifying title for the window title of the Exec.

FUNCTION is a function used to actually evaluate events, default is **EVAL-INPUT**.

ID is a string identifier to use for events generated in this Exec. *ID* defaults to the number given to the Exec process.

XCL:*PER-EXEC-VARIABLES* [Variable]

A list of pairs of the form (*VAR INIT*). Each time an Exec is entered, the variables in ***PER-EXEC-VARIABLES*** are rebound to the value returned by evaluating *INIT*. The initial value of ***PER-EXEC-VARIABLES*** is:

```
((*PACKAGE* *PACKAGE*)
 (* *)
 (** **)
 (*** ***)
 (+ +)
 (++ ++)
 (+++ +++)
 (- -)
 (/ /)
 (// //)
 (/// ///)
 (HELPFLAG T)
 (*EVALHOOK* NIL)
 (*APPLYHOOK* nil)
 (*ERROR-OUPUT* *TERMINAL-IO*)
 (*READTABLE* *READTABLE*)
```

```
(*package* *package*)
(*eval-function* *eval-function*)
(*exec-prompt* *exec-prompt*)
(*debugger-prompt* *debugger-prompt*))
```

Most of these cause the values to be (re)bound to their current value in any inferior Exec, or to **NIL**, their value at the "top level".

XCL:*EVAL-FUNCTION* [Variable]

Bound to the function used by the Exec to evaluate input. Typically in an INTERLISP Exec this is **IL:EVAL**, and in a Common Lisp Exec, **CL:EVAL**.

XCL:*EXEC-PROMPT* [Variable]

Bound to the string printed by the Exec as a prompt for input. Typically in an INTERLISP Exec this is "▮", and in a Common Lisp Exec, " > ".

XCL:*DEBUGGER-PROMPT* [Variable]

Bound to the string printed by the debugger Exec as a prompt for input. Typically in an INTERLISP Exec this is "▮ ", and in a Common Lisp Exec, ": ".

(XCL:EXEC-EVAL *FORM &OPTIONAL ENVIRONMENT*) [Function]

Evaluates *FORM* (using **EVAL**) in the lexical environment *ENVIRONMENT* the same as though it were typed in to **EXEC**, i.e., the event is recorded, and the evaluation is made undoable by substituting the UNDOABLE-functions for the corresponding destructive functions. **XCL:EXEC-EVAL** returns the value(s) of the form, but does not print it, and does not reset the variables *, **, ***, etc.

(XCL:EXEC-FORMAT *CONTROL-STRING &REST ARGUMENTS*) [Function]

In addition to saving inputs and values, the Exec saves many system messages on the history list. For example, **FILE CREATED** ..., *FN* **redefined**, *VAR* **reset**, output of **TIME**, **BREAKDOWN**, **ROOM**, save their output on the history list, so that when **??** prints the event, the output is also printed. The function **XCL:EXEC-FORMAT** can be used in user code similarly. **XCL:EXEC-FORMAT** performs (APPLY #'CL:FORMAT *TERMINAL-IO* *CONTROL-STRING ARGUMENTS*) and also saves the format string and arguments on the history list associated with the current event.

(XCL:SET-EXEC-TYPE *NAME*) [Function]

Sets the type of the current Exec to that indicated by *NAME*. This can be used to set up the Exec to your liking. *NAME* may be an atom or string. Possible names are:

INTERLISP, IL	*READTABLE* INTERLISP
	PACKAGE INTERLISP
	XCL:*DEBUGGER-PROMPT* "▮"
	XCL:*EXEC-PROMPT* "▮"
	XCL:*EVAL-FUNCTION* IL:EVAL
XEROX-COMMON-LISP, XCL	*READTABLE* XCL
	PACKAGE XCL-USER
	XCL:*DEBUGGER-PROMPT* ": "
	XCL:*EXEC-PROMPT* " > "
	XCL:*EVAL-FUNCTION* CL:EVAL
COMMON-LISP, CL	*READTABLE* LISP
	PACKAGE USER
	XCL:*DEBUGGER-PROMPT* ": "
	XCL:*EXEC-PROMPT* " > "
	XCL:*EVAL-FUNCTION* CL:EVAL
OLD-INTERLISP-T	*READTABLE* OLD-INTERLISP-T
	PACKAGE INTERLISP
	XCL:*DEBUGGER-PROMPT* "▮"
	XCL:*EXEC-PROMPT* ": "
	XCL:*EVAL-FUNCTION* IL:EVAL

(XCL:SET-DEFAULT-EXEC-TYPE *NAME*) [Function]

Like **XCL:SET-EXEC-TYPE** , but sets the type of Execs created by default, as from the background menu. Initially **XCL**. This can be used in your greet file to set default Execs to your liking.

Editing Exec Input

The Exec features an editor for input which provides completion, spelling correction, help facility, and character-level editing. The implementation is borrowed from the Interlisp module **TTYIN**. This section describes the use of the **TTYIN** editor from the perspective of the Exec.

Editing Your Input

Some editing operations can be performed using any of several characters; characters that are interrupts will, of course, not be read, so several alternatives are given. The following characters may be used to edit your input:

CONTROL-A, BACKSPACE — Deletes a character. At the start of the second or subsequent lines of your input, deletes the last character of the previous line.

CONTROL-W	Deletes a "word". Generally this means back to the last space or parenthesis.
CONTROL-Q	Deletes the current line, or if the current line is blank, deletes the previous line.
CONTROL-R	Refreshes the current line. Two in a row refreshes the whole buffer (when doing multiline input).
ESCAPE	Tries to complete the current word from the spelling list **USERWORDS**. In the case of ambiguity, completes as far as is uniquely determined, or beeps.
UNDO key (on 1108 and 1186) Middle-blank key (on 1132)	Retrieves characters from the previous non-empty buffer when it is able to; e.g., when typed at the beginning of the line this command restores the previous line you typed; when typed in the middle of a line fills in the remaining text from the old line; when typed following CONTROL-Q or CONTROL-W restores what those commands erased.
CONTROL-X	Goes to the end of your input (or end of expression if there is an excess right parenthesis) and returns if parentheses are balanced.

During most kinds of input, lines are broken, if possible, so that no word straddles the end of the line. The pseudo-carriage return ending the line is still read as a space, however; i.e., the program keeps track of whether a line ends in a carriage return or is merely broken at some convenient point. You will not get carriage returns in your strings unless you explicitly type them.

Using the Mouse

Editing with the mouse during TTYIN input is slightly different than with other modules. The mouse buttons are interpreted as follows during TTYIN input:

LEFT	Moves the caret to where the cursor is pointing. As you hold down *LEFT*, the caret moves around with the cursor; after you let up, any type-in will be inserted at the new position.
MIDDLE or *LEFT + RIGHT*	Like *LEFT*, but moves only to word boundaries.
RIGHT	*Deletes* text from the caret to the cursor, either forward or backward. While you hold down *RIGHT*, the text to be deleted is inverted; when you let up, the text goes away. If you let up outside the scope of the text, nothing is deleted (this is how to cancel this operation).

If you hold down *MOVE*, *COPY*, *SHIFT* or *CTRL* while pressing the mouse buttons, you instead get secondary selection, move selection or delete selection. The selection is made by holding the appropriate key down while pressing the mouse buttons *LEFT* (to select a character) or *MIDDLE* (to select a word), and optionally extend the selection either left or right using *RIGHT*. While you are doing this, the caret does not move, but the selected text is highlighted in a manner indicating what is about to happen. When the selection is complete, release the mouse buttons and then lift up on *MOVE/COPY/CTRL/SHIFT* and the appropriate action will occur:

COPY or *SHIFT* — The selected text is inserted as if it were typed. The text is highlighted with a broken underline during selection.

CTRL — The selected text is deleted. The text is complemented during selection.

MOVE or *CTRL + SHIFT* — Combines copy and delete. The selected text is moved to the caret.

You can cancel a selection in progress by pressing *LEFT* or *MIDDLE* as if to select, and moving outside the range of the text.

The most recent text deleted by mouse command can be inserted at the caret by typing the UNDO key (on the Xerox 1108/1186/1185) or the Middle-blank key (on the Xerox 1132). This is the same key that retrieves the previous buffer when issued at the end of a line.

Editing Commands

A number of characters have special effects while typing to the Exec. Some of them merely move the caret inside the input stream. While caret positioning can often be done more conveniently with the mouse, some of the commands, such as the case changing commands, can be useful for modifying the input.

In the descriptions below, current word means the word the cursor is under, or if under a space, the previous word. Currently, parentheses are treated as spaces, which is usually what you want, but can occasionally cause confusion in the word deletion commands. The notation [*CHAR*] means meta-*CHAR*. The notation $ stands for the ESCAPE/EXPAND key. Most commands can be preceded by numbers or escape (means infinity), only the first of which requires the meta key (or the edit prefix). Some commands also accept negative arguments, but some only look at the magnitude of the argument. Most of these commands are confined to work within one line of text unless otherwise noted.

Cursor Movement Commands

[bs] — Backs up one (or n) characters.

[space] — Moves forward one (or n) characters.

[↑] — Moves up one (or n) lines.

[lf] — Moves down one (or n) lines.

[(] — Moves back one (or n) words.

[)] — Moves ahead one (or n) words.

[tab] — Moves to end of line; with an argument moves to nth end of line; [$tab] goes to end of buffer.

[control-L] — Moves to start of line (or nth previous, or start of buffer).

[{] and [}] — Goes to start and end of buffer, respectively (like [$control-L] and [$tab]).

[[ ] (meta-left-bracket) Moves to beginning of the current list, where cursor is currently under an element of that list or its closing paren. (See also the auto-parenthesis-matching feature below under "Assorted Flags".)

[ ]] (meta-right-bracket) Moves to end of current list.

[Sx] Skips ahead to next (or nth) occurrence of character x, or rings the bell.

[Bx] Backward search, i.e., short for [-S] or [-nS].

Buffer Modification Commands

[Zx] Zaps characters from cursor to next (or nth) occurrence of x. There is no unzap command.

[A] or [R] Repeats the last S, B, or Z command, regardless of any intervening input.

[K] Kills the character under the cursor, or n chars starting at the cursor.

[cr] When the buffer is empty is the same as undo i.e. restores buffer's previous contents. Otherwise is just like a <cr> (except that it also terminates an insert). Thus, [<cr><cr>] will repeat the previous input (as will undo<cr> without the meta key).

[O] Does "Open line", inserting a crlf after the cursor, i.e., it breaks the line but leaves the cursor where it is.

[T] Transposes the characters before and after the cursor. When typed at the end of a line, transposes the previous two characters. Refuses to handle odd cases, such as tabs.

[G] Grabs the contents of the previous line from the cursor position onward. [nG] grabs the nth previous line.

[L] Puts the current word, or n words on line, in lower case. [$L] puts the rest of the linein lower case; or if given at the end of line puts the entire line in lower case.

[U] Analogous to [L], for putting word, line, or portion of line in upper case.

[C] Capitalizes. If you give it an argument, only the first word is capitalized; the rest are just lowercased.

[control-Q] Deletes the current line. [$control-Q] deletes from the current cursor position to the end of the buffer. No other arguments are handled.

[control-W] Deletes the current word, or the previous word if sitting on a space.

Miscellaneous Commands

[P] Prettyprints buffer. Clears the buffer and reprints it using prettyprint. If there are not enough right parentheses, it will supply more; if there are too many, any excess remains unprettyprinted at the end of the buffer. May refuse to do anything if there is an unclosed string or other error trying to read the buffer.

[N] Refreshes line. Same as control-R. [$N] refreshes the whole buffer; [nN] refreshes n lines. Cursor movement in TTYIN depends on TTYIN being the only source of output to the window; in some circumstances, you may need to refresh the line for best results.

[control-Y] Gets an Interlisp Exec.

[$control-Y] Gets an Interlisp Exec, but first unreads the contents of the buffer from the cursor onward. Thus if you typed at TTYIN something destined for Interlisp, you can do [control-L$control-Y] and give it to Lisp.

[←] Adds the current word to the spelling list **USERWORDS**. With zero argument, removes word. See **TTYINCOMPLETEFLG**.

Useful Macros

If the event is considered short enough, the Exec command **FIX** will load the buffer with the event's input, rather than calling the structure editor. If you really wanted the Lisp editor for your fix, you can say **FIX** *EVENT* - |**TTY**:|.

?= Handler

Typing the characters ?= <cr> displays the arguments to the function currently in progress. Since TTYIN wants you to be able to continue editing the buffer after a ?=, it prints the arguments below your type-in and then puts the cursor back where it was when ?= was typed.

Assorted Flags

These flags control aspects of TTYIN's behavior. Some have already been mentioned. In Interlisp-D, the flags are all initially set to **T**.

?ACTIVATEFLG [Variable]

If true, enables the feature whereby ? lists alternative completions from the current spelling list.

SHOWPARENFLG [Variable]

If true, then whenever you are typing Lisp input and type a right parenthesis, TTYIN will briefly move the cursor to the matching parenthesis, assuming it is still on the screen. The cursor stays there for about 1 second, or until you type another character (i.e., if you type fast you will never notice it).

USERWORDS [Variable]

USERWORDS contains words you mentioned recently: functions you have defined or edited, variables you have set or evaluated at the executive level, etc. This happens to be a very convenient list for context-free escape completion; if you have recently

edited a function, chances are good you may want to edit it again (typing "ED(xx$)") or type a call to it. If there is no completion for the current word from **USERWORDS**, or there is more than one possible completion, TTYIN beeps. If typed when not inside a word, Escape completes to the value of **LASTWORD**, i.e., the last thing you typed that the Exec noticed, except that Escape at the beginning of the line is left alone (it is an Old Interlisp Exec command).

If you really wanted to enter an escape, you can, of course, just quote it with a CONTROL-V, like you can other control characters.

You may explicitly add words to **USERWORDS** yourself that would not get there otherwise. To make this convenient online the edit command [←] means "add the current atom to **USERWORDS**" (you might think of the command as pointing out this atom). For example, you might be entering a function definition and want to point to one or more of its arguments or prog variables. Giving an argument of zero to this command will instead remove the indicated atom from **USERWORDS**.

Note that this feature loses some of its value if the spelling list is too long, if there are too many alternative completions for you to get by with typing a few characters followed by escape. Lisp's maintenance of the spelling list **USERWORDS** keeps the temporary section (which is where everything goes initially unless you say otherwise) limited to **\#USERWORDS** atoms, initially 100. Words fall off the end if they haven't been used (they are used if **FIXSPELL** corrects to one, or you use <escape> to complete one).

[This page intentionally left blank]

APPENDIX B. SEDIT—THE LISP EDITOR

SEdit is the new Xerox Lisp structure editor. It allows you to edit Xerox Lisp code directly in memory. This editor replaces DEdit in Chapter 16, Structure Editor, of the *Interlisp-D Reference Manual*.

16.1 SEdit - The Structure Editor

As a structure editor, SEdit alters Lisp code directly in memory. The effect this has on the running system depends on what is being edited.

For Common Lisp definitions, SEdit always edits a copy of the object. For example, with functions, it edits the definition of the function. What the system actually runs is the installed function, either compiled or interpreted. The primary difference between the definition and the installed function is that comment forms are removed from the definition to produce the installed function. The changes made while editing a function will not be installed until the edit session is complete.

For Interlisp functions and macros, SEdit edits the actual structure that will be run. An exception to this is an edit of an EXPR definition of a compiled function. In this case, changes are included and the function is unsaved when the edit session is completed.

SEdit edits all other structures, such as variables and property lists, directly. SEdit installs all changes as they are made.

If an error is made during an SEdit session, abort the edit with an Abort command (see Section 16.1.7, Command Keys). This command undoes all changes from the beginning of the edit session and exits from SEdit without changing your environment.

If the definition being edited is redefined while the edit window is open, SEdit redisplays the new definition. Any edits on the old definition will be lost. If SEdit was busy when the redefinition occurred, the SEdit window will be gray. When SEdit is no longer busy, position the cursor in the SEdit window and press the left mouse button; SEdit will get the new definition and display it.

16.1.1 An Edit Session

The List Structure Editor discussion in Chapter 3, Language Integration, explains how to start an editor in Xerox Lisp.

Whenever you call SEdit, a new SEdit window is created. This SEdit window has its own process, and thus does not rely on an

Exec to run in. You can make edits in the window, shrink it while you do something else, expand it and edit some more, and finally close the window when you are done.

Throughout an edit session, SEdit remembers everything that you do through a change history. All edits can be undone and redone sequentially. When an edit session ends, SEdit forgets this information and installs the changes in the system.

The session ends with an event signalling to the editor that changes are complete. Three events signal completion:

- Closing the window.

Do this to terminate the edit session when you are finished.

- Shrinking the window.

Shrink the window when you have made some edits and want to continue the editing session at a later time.

- Typing CONTROL-X.

Use this command when you want to install your changes and complete the edit. CONTROL-X leaves the edit window open and ready for more editing while the TTY process passes back to the Exec.

A new edit session begins when you come back to an SEdit after shrinking or using CONTROL-X. The change history is discarded at this point.

If the Exec is waiting for SEdit to return before going on, complete the edit session using any of the methods above to alert the Exec that SEdit is done. The TTY process passes back to the Exec.

16.1.2 SEdit Carets

There are two carets in SEdit, the edit caret and the structure caret. The edit caret appears when characters are edited within a single structure, such as an atom, string, or comment. Anything typed in will appear at the edit caret as part of the structure that the caret is within. The edit caret looks like this:

The structure caret appears when the edit point is between structures, so that anything inserted will go into a new structure. It looks like this:

SEdit changes the caret frequently, depending on where you are in the structure you are editing, and how the caret is positioned. The left mouse button allows an edit caret position to be set. The middle mouse button allows the structure caret position to be set.

16.1.3 The Mouse

In SEdit, the mouse buttons are used as follows. The left mouse button positions the mouse cursor to point to parts of Lisp structures. The middle mouse button positions the mouse cursor to point to whole Lisp structures. Thus, selecting the Q in LEQ using the left mouse button selects that character, and sets the edit caret after the Q:

Any characters typed in at this point would be appended to the atom LEQ.

Selecting the same letter using the middle mouse button selects the whole atom (this convention matches TEdit's character/word selection convention), and sets a structure caret between the LEQ and the n:

At this point, any characters typed in would form a new atom between the LEQ and the n.

Larger structures can be selected in two ways. Use the middle mouse button to position the mouse cursor on the parenthesis of the desired list to select that list. Press the mouse button multiple times, without moving the mouse, extends the selection. Using the previous example, if the middle button were pressed twice, the list (LEQ ...) would be selected:

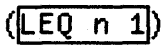
Pressing the button a third time would cause the list containing the (LEQ n 1) to be selected.

The right mouse button positions the mouse cursor for selecting sequences of structures or substructures. Extended selections are indicated by a box enclosing the structures selected. The selection is extended in the same mode as the original selection. That is, if the original selection were a character selection, the right button could be used to select more characters in the same atom. Extended selections also have the property of being marked for pending deletion. That is, the selection takes the place of the caret, and anything typed in is inserted in place of the selection.

For example, selecting the E by pressing the left mouse button and selecting the Q by pressing the right mouse button would produce:

Similarly, pressing the middle mouse button and then selecting with the right mouse button extends the selection by whole structures. Thus, in our example, pressing the middle mouse

button to select LEQ and pressing the right mouse button to select the 1 would produce:



This is not the same as selecting the entire list, as above. Instead, the elements in the list are collectively selected, but the list itself is not.

16.1.4 Gaps

The SEdit structure editor requires that everything edited must have an underlying Lisp structure, even if the structure is not directly displayed. For example, with quoted forms the actual structure might be **(QUOTE** GREEN), although this would be displayed as **'GREEN**. Even when the user is in the midst of typing in a form, the underlying Lisp structure must exist.

Because of this necessity, SEdit provides gaps to serve as dummy Lisp objects during typing. SEdit does not need a gap for every form typed in, but gaps are necessary for quoted objects. When something is typed that requires SEdit to build a Lisp structure and thus create a gap, as the quote character does, the gap will appear marked for pending deletion. This means it is ready to be replaced by the structure to be typed in. In this way it is possible to type special structures, like quotes, directly, while SEdit maintains the structure.

A gap looks like: -x-

A gap displayed after a quote has been typed in would look like this:

with the gap marked for pending deletion, ready for typein of the object to be quoted.

16.1.5 Special Characters

A few characters have special meaning in Lisp, and are treated specially by SEdit. SEdit must always have a complete structure to work on at any level of the edit. This means that SEdit needs a special way to type in structures such as lists, strings, and quoted objects. In most instances these structures can be typed in just as they would be to a regular Exec, but in a few cases this is not possible.

Lists- (and) Lists begin with an open parenthesis character (. Typing an open parenthesis gives a balanced list, that is, SEdit inserts both an open and a close parenthesis. The structure caret is between the two parentheses. List elements can be typed in at the structure caret. When a close parenthesis,) is typed, the caret will be moved outside the list (and the close parenthesis), effectively finishing the list.

Quoted Structures: SEdit handles the quote keys so that it is possible to type in all quote forms directly. When typing one of the following quote

keys at a structure caret, the quote character typed will appear, followed by a gap to be replaced by the object to be quoted.

Single Quote – ' Use to enter quoted structures.

Backquote –' Use to enter backquoted structures.

Comma – , Use to enter comma forms, as used with a Backquote form.

At Sign – @ Use after a comma to create a comma-at-sign gap. This allows type-in of comma-at forms, e.g. **,@list**, as used within a Backquote form.

Dot – . Use the dot (period) after a comma to create a comma-dot gap. This allows type-in of comma-dot forms, e.g. **, .list**, as used within a Backquote form.

Meta-# or Meta-3 Use to enter the **CL:FUNCTION** abbreviation hash–quote (#'). A hash-quote gap will follow typein of Meta-#.

Dotted Lists: The dot, or period, character (.) is used to type dotted lists in SEdit. After typing a dot, SEdit inserts a dot and a gap to fill in for the tail of the list. To dot an existing list, point the cursor between the last and second to the last element in the list, and type a dot. To undot a list, select the tail of the list before the dot while holding down the SHIFT key.

Escape- \ or % Use to escape from a specific typed in character. Use the escape key to enter characters, like parentheses, which otherwise have special meaning to the SEdit reader. Press the escape key then type in the character to escape. SEdit uses the escape key appropriate to the environment it is editing in; it depends on the readtable that was current when the editor was started. The backslash key (\) is used when editing Common Lisp, and the percent key (%) is used when editing Interlisp.

Multiple Escape- | Use the multiple escape key, the vertical bar character (|), to escape a sequence of typed in characters. SEdit always balances multiple escape characters. When one multiple escape character is typed, SEdit produces a balanced pair, with the caret between them, ready for typing in the characters to be escaped. If you type a second vertical bar, the caret moves after the second vertical bar, and is still within the same atom, so that you can add more unescaped characters to the atom.

Comments- ; The comment key, a semicolon (;), starts a comment. When a semicolon is typed, an empty comment is inserted with the caret in position for typing in the comment. Comments can be edited like strings. There are three levels of comments supported by SEdit: single, double, and triple. Single semicolon comments are formatted at the comment column, about three-quarters of the way across the SEdit window, towards the right margin. Double semicolon comments are formatted at the current indentation of the code that they are in. Triple semicolon comments are formatted against the left margin of the SEdit window. The level of a comment can be increased or decreased by pointing after the semicolon, and either typing another semicolon, or backspacing over the preceding semicolon. Comments can be placed anywhere in your Common Lisp code. However, in

Interlisp code, they must follow the placement rules for Interlisp comments.

Strings- " Enter strings in SEdit by typing a double quote ("). SEdit balances the double quotes. When one is typed, SEdit produces a second, with the caret between the two, ready for typing the characters of the string. If a double quote character is typed in the middle of a string, SEdit breaks the string into two smaller strings, leaving the caret between them.

16.1.6 Control Keys

SEdit uses Control Keys for certain simple editing operations.

CONTROL-L [Editor Command]

Redisplays the structure being edited.

CONTROL-W [Editor Command]

Deletes the previous atom or whole structure. If the caret is in the middle of an atom, deletes backward to the beginning of the atom only.

CONTROL-X [Editor Command]

Signals the system that this edit is complete. The window remains open, though, so the user can see the edit and start editing again directly.

16.1.7 Command Keys

SEdit commands are most easily entered through the keyboard. They are all single character META keystrokes. On 1108s the Meta key is labelled OPEN; on 1186s it is labelled META (ALT).

For all alphabetic command keys, either uppercase or lowercase works. There is also an attached menu available, described in detail in Section 16.9, SEdit Command Menu.

Meta- (or Meta-9 [Editor Command]

Parenthesizes the current selection, positioning the caret at the beginning of the new list. Only a whole structure selection or an extended selection of a sequence of whole structures can be parenthesized.

Meta-) or Meta-0 [Editor Command]

Parenthesizes the current selection, positioning the caret after the new list.

Meta-' Meta-' Meta-, Meta-@ or Meta-2, Meta-. [Editor Command]

Quotes the current selection with the specified kind of quote, respectively, Single Quote, Backquote, Comma, Comma-At-Sign, or Comma-Dot.

Meta- / [Editor Command]

Extracts one level of structure from the current selection. If the current selection is an atom, or if there is no selection, the next largest structure containing this atom, or caret, is used. This command can be used to strip the parentheses off a list or a comment, or to unquote a quoted structure.

Meta-; [Editor Command]

Converts old style comments in the selected structure to new style comments. This converter notices any list that begins with an asterisk (*) in the INTERLISP package (IL:*) as an old style comment. Section 16.1.11, Options, describes the converter options .

Meta-A [Editor Command]

Aborts. This command must be confirmed. All changes since the beginning of the edit session are undone, and the edit is closed.

Meta-B [Editor Command]

Changes Print Base. Prompts for entry of the desired Print Base, in decimal. SEdit redisplays fixed point numbers in this new base.

Meta-E or Do-It [Editor Command]

Evaluates the current selection. If the result is a structure, the inspector is called on it, allowing the user to choose how to look at the result. Otherwise, the result is printed in the SEdit prompt window. The evaluation is done in the process from which the edit session was started. Thus, while editing a function from a break window, evaluations are done in the context of the break.

Meta-F or FIND [Editor Command]

Finds a specified structure. If there is a current selection, SEdit looks for the next occurrence of the selected structure. If there is no selection, SEdit prompts for the structure to find, and searches forward from the position of the caret. The found structure will be selected, so the Find command can be used to easily find the same structure again.

Meta-H or HELP [Editor Command]

Shows the argument list for the function currently selected, or currently being typed in, in the SEdit prompt window. If the argument list will not fit in the SEdit prompt window, it is displayed in the main Prompt Window.

Meta-J [Editor Command]

Joins. This command joins any number of sequential Lisp objects of the same type into one object of that type. Join is supported for atoms, strings, lists, and comments. In addition, SEdit permits joining of a sequence of atoms and strings, since either type can easily be coerced into the other. In this case, the result of the Join will be an atom if the first object in the selection is an atom, otherwise the result will be a string.

Meta-M [Editor Command]

Attaches a menu of the commonly used commands (the SEdit Command Menu) to the top of the SEdit window. Each SEdit window can have its own menu, if desired.

Meta-N or SKIP-NEXT [Editor Command]

Skips to the next gap in the structure, leaving it selected for pending deletion.

Meta-O [Editor Command]

Edits the definition of the current selection. If the selected name has more than one type of definition, SEdit asks for the type to be edited. If the selection has no definition, a menu pops up. This menu lets the user specify either the type of definition to be created, or no definition if none needs to be created.

Meta-P [Editor Command]

Changes the current package for this edit. Prompts the user, in the SEdit prompt window, for a new package name. SEdit will redisplay atoms with respect to that package.

Meta-R or AGAIN [Editor Command]

Redoes the edit change that was just undone. Redo only works directly following an Undo. Any number of Undo commands can be sequentially redone.

Meta-S or SHIFT-FIND [Editor Command]

Substitutes one structure for another over the current selection. SEdit prompts the user in the SEdit prompt window for the structure to replace, and the structure to replace it with.

Meta-U or UNDO [Editor Command]

Undoes the last edit. All changes since the beginning of the edit session are remembered, and can be undone sequentially.

Meta-X or EXPAND [Editor Command]

Replaces the current selection with its definition. This command can be used to expand macros and translate CLISP.

Meta-Z [Editor Command]

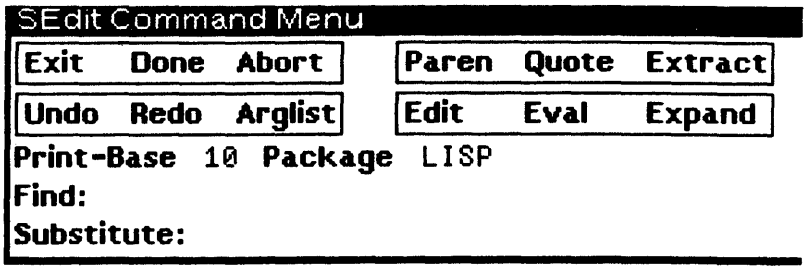
Mutates. This command allows the user to do arbitrary operations on a LISP structure. First select the structure to be mutated (it must be a whole structure, not an extended selection). When the user presses Meta-Z SEdit prompts for the function to use for mutating. This function is called with the selected structure as its argument, and the structure is replaced with the result of the mutation.

For example, an atom can be put in upper case by selecting the atom and mutating by the function U-CASE. You can replace a structure with its value by selecting it and mutating by EVAL.

16.1.8 Command Menu

The SEdit Attached Command Menu contains the commonly used commands. Use the Meta-M keyboard command to bring

up this menu. The menu can be closed, independently of the SEdit window, when desired. The menu looks like:



All of the commands in the menu function identically to their corresponding keyboard commands, except for Find and Substitute.

When Find is selected with the mouse cursor, SEdit prompts in the menu window, next to the Find button, for the expression to find. Type in the expression then select Find again. The search begins from the caret position in the SEdit window.

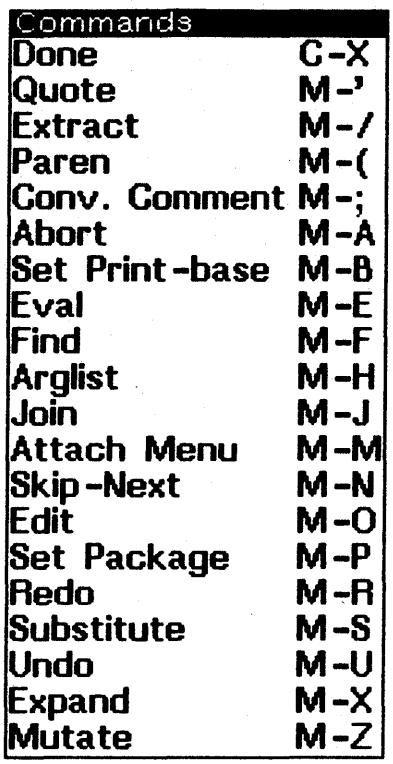
Similarly, Substitute prompts, next to the Find button, for the expression to find, and next to the Substitute button for the expression to substitute it with. After both expressions have been typed in, selecting Substitute replaces all occurrences of the Find expression in the current selection with the Substitute expression.

To do a confirmed substitute, set the edit point before the first desired substitution, and select Find. Then if you want to substitute that occurrence of the expression, select Substitute. Otherwise, select Find again to go on.

Selecting either Find or Substitute with the right mouse button erases the old structure to find or substitute from the menu, and prompts for a new one.

16.1.9 Help Menu

When the mouse cursor is positioned in the SEdit title bar and the middle mouse button is pressed, a Help Menu of commands pops up. The menu looks like this:



The Help Menu lists each command and its corresponding Command Key. (In the menu, the letter C stands for CONTROL, while M indicates Meta.) The command selected is executed just as if the command had been entered from the keyboard. The menu remembers which command was selected last, and pops up with the mouse cursor next to that same command the next time the menu is used. This provides a very fast way to repeat the same command when using the mouse.

16.1.10 Interface

SEdit has specific functions which allow the user to control certain aspects of Sedit's behavior.

(SEDIT.GET.WINDOW.REGION *CONTEXT REASON***)** [Function]

This function is called when SEdit wants to know where to place a window it is about to open. This happens whenever the user starts a new SEdit or expands an Sedit icon.The default behavior is to pop a window region off SEdit's stack of regions that have been used in the past. If the stack is empty, SEdit prompts for a new region.

This function can be redefined to provide different behavior. It is called with the edit *CONTEXT* and a *REASON* for needing a region. The edit *CONTEXT* is SEdit's main data structure and can be useful for associating particular edits with specific regions. The *REASON* argument specifies why SEdit wants a region, with one of the keywords, :CREATE or :EXPAND.

(SEDIT.SAVE.WINDOW.REGION *CONTEXT REASON*) [Function]

This function is called whenever SEdit is finished with a region and wants to make the region available for other SEdits. This happens whenever an SEdit window is closed or shrunk. The default behavior is simply to push the region onto SEdit's stack of regions (see the **SEDIT.KEEP.WINDOW.REGION** option in Section 16.1.11, Options).

This function can be redefined to provide different behavior. It is also called with the edit *CONTEXT*,used for associating particular edits with specific regions, and *REASON*. The *REASON* argument specifies why the region should be saved the region; it is one of the keywords :CLOSE or :SHRINK.

(SEDIT.RESET) [Function]

This function recomputes the SEdit edit environment. Any changes made in the font profile, or any changes made to SEdit's commands are captured by resetting. Close all SEdit windows before calling this function.

16.1.11 Options

The following top level variables can be set as desired:

SEDIT.KEEP.WINDOW.REGION [Variable]

Default **T**. This flag determines the behavior of the default SEdit region manager, explained above, for shrinking and expanding windows. When **T**, shrinking an SEdit window will not give up that window's region; the icon will always expand back into the same region. When **NIL**, the window's region is made available for other SEdits when the window is shrunk. Then when an SEdit icon is expanded, the window will be reshaped to the next available region.

This variable is only used by the default implementations of the functions **SEDIT.GET.WINDOW.REGION** and **SEDIT.SAVE.WINDOW.REGION**. If these functions are redefined, this flag is no longer used.

SEDIT.CONVERT.UPGRADE [Variable]

Default 100. When using Meta-; to convert old-style single-asterisk comments, if the length of the comment exceeds **SEDIT.CONVERT.UPGRADE** characters, the comment is converted into a double semicolon comment. Otherwise, the comment is converted into a single semicolon comment.

Old-style double-asterisk comments are always converted into new-style triple-semicolon comments.

[This page intentionally left blank]

ICONW, used to build small windows which will appear as icons on the display, is a new standard input/output feature of Xerox Lisp. The following description of **ICONW** should be appended to Section 28.4, Windows, of the *Interlisp-D Reference Manual*.

28.4.16 Creating Icons with ICONW

ICONW is a group of functions available for building small windows of arbitrary shape. These windows are principally for use as icons for shrinking windows; i.e., these functions are likely to be invoked from within the **ICONFN** of a window. An icon is specified by supplying its image (a bitmap) and a mask that specifies its shape. The mask is a bitmap of the same dimensions as the image whose bits are on (black) in those positions considered to be in the image, and off (white) in those positions where the background should show through. By using the mask and appropriate window functions, **ICONW** maintains the illusion that the icon window is nonrectangular, even though the actual window itself is rectangular. The illusion is not complete, of course. For example, if you try to select what looks like the background (or an occluded window) around the icon but still within its rectangular perimeter, the icon window itself is selected. Also, if you move a window occluded by an icon, the icon never notices that the background changed behind it. Icons created with **ICONW** can also have titles; some part of the image can be filled with text computed at the time the icon is created, or text may be changed after creation.

28.4.16.1 Creating Icons

Two types of icons can be created with **ICONW**, a borderless window containing an image defined by a mask and a window with a title.

(ICONW *IMAGE MASK POSITION NOOPENFLG***)** [Function]

Creates a window at *POSITION*, or prompts for a position if *POSITION* is **NIL**. The window is borderless, and filled with *IMAGE*, as cookie-cut by *MASK*. If *MASK* is **NIL**, the image is considered rectangular (i.e., *MASK* defaults to a black bitmap of the same dimensions as *IMAGE*). If *NOOPENFLG* is **T**, the window is returned unopened.

(TITLEDICONW *ICON TITLE FONT POSITION NOOPENFLG JUST BREAKCHARS OPERATION***)** [Function]

Creates a titled icon at *POSITION*, or prompts for a position if *POSITION* is **NIL**. If *NOOPENFLG* is **T**, the window is returned unopened. The argument *ICON* is an instance of the record **TITLEDICON**, which specifies the icon image and mask, as with **ICONW**, and a region within the image to be used for displaying the title. Thus, the *ICON* argument is usually of the form

```
(create  TITLEDICON  ICON ← someIconImage
MASK ← iconMask  TITLEREG ←
someRegionWithinICON)
```

The title region is specified in coordinates relative to the icon, i.e., the lower-left corner of the image bitmap is (0, 0). The mask can be **NIL** if the icon is rectangular. The image should be white where it is covered by the title region. **TITLEDICONW** clears the region before printing on it. The title is printed into the specified region in the image, using *FONT*. If *FONT* is **NIL** it defaults to the value of **DEFAULTICONFONT**, initially Helvetica 10. The title is broken into multiple lines if necessary; **TITLEDICONW** attempts to place the breaks at characters that are in the list of character codes *BREAKCHARS*. *BREAKCHARS* defaults to **(CHARCODE (SPACE –))**. In addition, line breaks are forced by any carriage returns in *TITLE*, independent of *BREAKCHARS*. *BREAKCHARS* is ignored if a long title would not otherwise fit in the specified region. For convenience, *BREAKCHARS* = **FILE** means the title is a file name, so break at file name field delimiters. The argument *JUST* indicates how the text should be justified relative to the region. It is an atom or list of atoms chosen from **TOP**, **BOTTOM**, **LEFT**, or **RIGHT**, which indicate the vertical positioning (flush to top or bottom) and/or horizontal positioning (flush to left edge or right). If *JUST* = **NIL**, the text is centered. The argument *OPERATION* is a display stream operation indicating how the title should be printed. If *OPERATION* is **INVERT**, then the title is printed white-on-black. The default *OPERATION* is **REPLACE**, meaning black-on-white. **ERASE** is the same as **INVERT**; **PAINT** is the same as **REPLACE**.

For convenience, **TITLEDICONW** can also be used to create icons that consist solely of a title, with no special image. If the argument *ICON* is **NIL**, **TITLEDICONW** creates a rectangular icon large enough to contain *TITLE*, with a border the same width as that on a regular window. The remaining arguments are as described above, except that a *JUST* of **TOP** or **BOTTOM** is not meaningful.

28.4.16.2 Modifying Icons

(ICONW.TITLE *ICON TITLE***)** [Function]

Returns the current title of the window *ICON*, which must be a window returned by **TITLEDICONW**. In addition, if *TITLE* is non-**NIL**, makes *TITLE* the new title of the window and repaints it accordingly. To erase the current title, make *TITLE* a null string.

(ICONW.SHADE *WINDOW SHADE***)** [Function]

Returns the current shading of the window **ICON**, which must be a window returned by **ICONW** or **TITLEDICONW**. In addition, if *SHADE* is non-**NIL**, paints the texture *SHADE* on *WINDOW*. A typical use for this function is to communicate a change of state in a window that is shrunken, without reopening the window. To remove any shading, make *SHADE* be **WHITESHADE**.

28.4.16.3 Default Icons

When you shrink a window that has no **ICONFN**, the system currently creates an icon that looks like the window's title bar. You can make the system instead create titled icons by setting the global variable **DEFAULTICONFN** to the value **TEXTICON**.

(TEXTICON *WINDOW TEXT*) [Function]

Creates a titled icon window for the main window *WINDOW* containing the text *TEXT*, or the window's title if *TEXT* is **NIL**.

DEFAULTTEXTICON [Variable]

The value that **TEXTICON** passes to **TITLEDICONW** as its *ICON* argument. Initially it is **NIL**, which creates an unadorned rectangular window. However, you can set it to a **TITLEDICON** record of your choosing if you would like default icons to have a different appearance.

28.4.16.4 Sample Icons

The LispUsers StockIcons module contains a collection of icons and their masks usable with **ICONW**, including:

- **FOLDER**, **FOLDERMASK** - a file folder
- **PAPERICON**, **PAPERICONMASK** - a sheet of paper with the top right corner turned
- **FILEDRAWER**, **FILEDRAWERMASK** - front of a file drawer
- **ENVELOPEICON**, **ENVELOPEMASK** - envelope
- **TITLED.FILEDRAWER** - TitledIcon of the filedrawer front (capacity, about three lines of 10-point text)
- **TITLED.FILEFOLDER** - TitledIcon of the file folder (capacity, about three lines of 10-point text)
- **TITLED.ENVELOPE** - TitledIcon of the envelope (capacity, one short line of 10-point text)

[This page intentionally left blank]

APPENDIX D. Free Menu

Free Menu is a standard input/output feature of Xerox Lisp. The following description of **Free Menu** should be added to the Menus segment of Chapter 28, Windows and Menus, in the *Interlisp-D Reference Manual*.

28.7 Free Menus

Free Menus are powerful and flexible menus that are useful for applications needing menus with different types of items, including command items, state items, and items that can be edited. A Free Menu is part of a window. It can can be opened and closed as desired, or attached as a control menu to the application window.

28.7.1 Making a Free Menu

A Free Menu is built from a description of the contents and layout of the menu. As a Free Menu is simply a group of items, a Free Menu Description is simply a specification of a group of items. Each group has properties associated with it, as does each Free Menu Item. These properties specify the format of the items in the group, and the behavior of each item. The function **FREEMENU** takes a Free Menu Description, and returns a closed window with the Free Menu in it.

The easiest way to make a Free Menu is to define a specific function which calls **FREEMENU** with the Free Menu Description in the function. This function can then also set up the Free Menu window as required by the application. The Free Menu Description is saved as part of the specific function when the application is saved. Alternately, the Free Menu Description can be saved as a variable in your file; then just call **FREEMENU** with the name of the variable. This may be a more difficult alternative if the backquote facility is used to build the Free Menu Description (see Section 28.7.7, Free Menu Item Descriptions, for more information on using backquote with a Free Menu Description) .

28.7.2 Free Menu Formatting

A Free Menu can be formatted in one of four ways. The items in any group can be automatically laid out in rows, in columns, or in a table, or else the application can specify the exact location of each item in the group. Free Menu keeps track of the region that a group of items occupies, and items can be justified within that region. This way an item can be automatically positioned at one of the nine justification locations, top-left, top-center, top-right, middle-left, etc.

28.7.3 Free Menu Description

A Free Menu Description, specifying a group of items, is a list structure. The first entry in the list is an optional list of the properties for this group of items. This entry is in the form:

(PROPS <PROP> <VALUE> <PROP> <VALUE> ...)

The keyword **PROPS** determines whether or not the optional group properties list is specified. Section 28.7.4, "Free Menu Group Properties, " describes each group property.

One important group property is **FORMAT**. The four types of formatting, **ROW, TABLE, COLUMN**, or **EXPLICIT**, determine the syntax of the rest of the Free Menu Description. When using **EXPLICIT** formatting, the rest of the description is any number of Item Descriptions which have **LEFT** and **BOTTOM** properties specifying the position of the item in the menu. The syntax is:

```
((PROPS FORMAT EXPLICIT ...)
 <ITEM DESCRIPTION>
 <ITEM DESCRIPTION> ...)
```

When using **ROW** or **TABLE** formatting, the rest of the description is any number of item groups, each group corresponding to a row in the menu. These groups are identical in syntax to an **EXPLICIT** group description. The groups have an optional PROPS list and any number of Item Descriptions. The items need not have **LEFT** and **BOTTOM** properties, as the location of each item is determined by the formatter. However, the order of the rows and items is important. The menu is laid out top to bottom by row, and left to right within each row. The syntax is:

```
((PROPS FORMAT ROW ...)          ; props of this group
 (<ITEM DESCRIPTION>             ; items in first row
  <ITEM DESCRIPTION> ...)
 ((PROPS ...)                    ; props of second row
  <ITEM DESCRIPTION>             ; items in second row
  <ITEM DESCRIPTION> ...))
```

(The comments above only describe the syntax.)

For **COLUMN** formatting, the syntax is identical to that of **ROW** formatting. However, each group of items corresponds to a column in the menu, rather than a row. The menu is laid out left to right by column, top to bottom within each column.

Finally, a Free Menu Description can have recursively nested groups. Anywhere the description can take an Item Description, it can take a group, marked by the keyword **GROUP**. A nested group inherits all of the properties of its mother group, by default. However, any of these properties can be overridden in the nested groups **PROPS** list, including the **FORMAT**. The syntax is:

```
(                       ; no PROPS list, default row format
(<ITEM DESCRIPTION>     ; first in row
(GROUP                  ; nested group, second in row
  (PROPS FORMAT COLUMN ...)   ; optional props
  (<ITEM DESCRIPTION> ...)    ; first column
  (<ITEM DESCRIPTION> ...))
   <ITEM DESCRIPTION>))       ; third in row
```

Here is an example of a simple Free Menu Description for a menu which might provide access to a simple data base:

```
(((LABEL LOOKUP SELECTEDFN MYLOOKUPFN)
  (LABEL EXIT SELECTEDFN MYEXITFN))
 ((LABEL Name: TYPE DISPLAY) (LABEL "" TYPE EDIT ID NAME))
 ((LABEL Address: TYPE DISPLAY) (LABEL "" TYPE EDIT ID ADDRESS))
 ((LABEL Phone: TYPE DISPLAY)
  (LABEL "" TYPE EDIT LIMITCHARS MYPHONEP ID PHONE)))
```

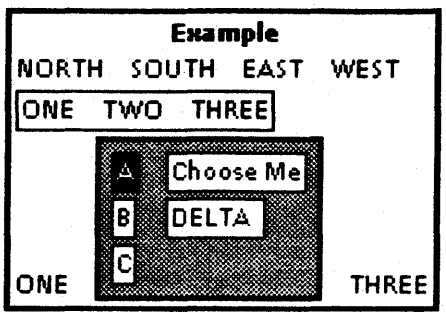
This menu has two command buttons, LOOKUP and EXIT, and three edit fields, with IDs NAME, PHONE, and ADDRESS. The Edit items are initialized to the empty string, as in this example they need no other initial value. The user could select the Name: prompt, type a person's name, and then press the LOOKUP button. The function **MYLOOKUPFN** would be called. That function would look at the NAME Edit item, look up that name in the data base, and fill in the rest of the fields appropriately. The PHONE item has **MYPHONEP** as a **LIMITCHARS** function. This function would be called when editing the phone number, in order to restrict input to a valid phone number. After looking up Perry, the Free Menu might look like:

```
LOOKUP  EXIT
Name:  Herbert Q Perry
Address:  13 Middleperry Dr
Phone:  (411) 767-1234
```

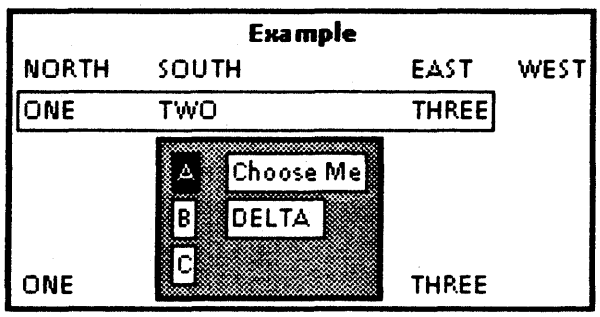
Here is a more complicated example:

```
((PROPS FONT (MODERN 10))
 ((LABEL Example FONT (MODERN 10 BOLD) HJUSTIFY CENTER))
 ((LABEL NORTH) (LABEL SOUTH) (LABEL EAST) (LABEL WEST))
 ((PROPS ID ROW3 BOX 1)
  (LABEL ONE) (LABEL TWO) (LABEL THREE))
 ((PROPS ID ROW4)
  (LABEL ONE ID ALPHA)
  (GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
    ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
     (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
     (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
   ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
          INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
    (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
  (LABEL THREE)))
```

which will produce the following Free Menu:



And if the Free Menu were formatted as a Table, instead of in Rows, it would look like:



The following breakdown of the example explains how each part contributes to the Free Menu shown above.

```
(PROPS FONT (MODERN 10))
```

This line specifies the properties of the group that is the entire Free Menu. These properties are described in Section 28.7.4, Free Menu Group Properties. In this example, all items in the Free Menu, unless otherwise specified, will be in Modern 10.

```
((LABEL Example FONT (MODERN 10 BOLD) HJUSTIFY CENTER))
```

This line of the Free Menu Description describes the first row of the menu. Since the **FORMAT** specification of a Free Menu is, by default, **ROW** formatting, this line sets the first row in the menu. If the menu were in **COLUMN** formatting, this position in the description would specify the first column in the menu.

In this example the first row contains only one item. The item is, by default, a type **MOMENTARY** item. It has its own Font declaration (FONT (MODERN 10 BOLD)), that overrides the font specified for the Free Menu as a whole, so the item appears bolded.

Finally, the item is justified, in this case centered. The **HJUSTIFY** Item Property indicates that the item is to be centered horizontally within its row.

```
((LABEL NORTH) (LABEL SOUTH) (LABEL EAST) (LABEL WEST))
```

This line specifies the second row of the menu. The second row has four very simple items, labeled NORTH, SOUTH, EAST, and WEST next to each other within the same row.

```
((PROPS ID ROW3 BOX 1)
 (LABEL ONE) (LABEL TWO) (LABEL THREE))
```

The third row in the menu is similar to the second row, except that it has a box drawn around it. The box is specified in the PROPS declaration for this row. Rows (and columns) are just like Groups in that the first thing in the declaration can be a list of properties for that row. In this case the row is named by giving it an ID property of ROW3. It is useful to name your groups if you want to be able to access and modify their properties later (via the function **FM.GROUPPROP**). It is boxed by specifying the **BOX** property with a value of 1, meaning draw the box one dot wide.

```
((PROPS ID ROW4)
 (LABEL ONE ID ALPHA)
 (GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
  ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
       (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
       (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
      ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
        INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
       (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
 (LABEL THREE)))
```

This part of the description specifies the fourth row in the menu. This row consists of: an item labelled ONE, a group of items, and an item labelled THREE. That is, Free Menu thinks of the group as an entry, and formats the rest of the row just as it it were a large item.

```
(GROUP (PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4)
 ((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
  (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
  (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
 ((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
   INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
  (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)))
```

The second part of this row is a nested group of items. It is declared as a group by placing the keyword GROUP as the first word in the declaration. A group can be declared anywhere a Free Menu Description can take a Free Menu Item Description (as opposed to a row or column declaration).

The first thing in what would have been the second item declaration in this row is the keyword GROUP. Following this keyword comes a normal group description, starting with an optional list of properties, and followed by any number of things to go in the group (based on the format of the group).

This group's Props declaration is:

```
(PROPS FORMAT COLUMN BACKGROUND 23130 BOX 2 BOXSPACE 4).
```

It specifies that the group is to be formatted as a number of columns (instead of rows, the default). The entire group will have a background shade of 23130, and a box of width 2 around it, as you can see in the sample menu. The BOXSPACE declaration tells Free Menu to leave an extra four dots of room between the edge of the group (ie the box around the group) and the items in the group.

The first column of this group is a Collection of **NWAY** items:

```
((TYPE NWAY LABEL A BOX 1 COLLECTION COL1 NWAYPROPS (DESELECT T))
 (TYPE NWAY LABEL B BOX 1 COLLECTION COL1)
 (TYPE NWAY LABEL C BOX 1 COLLECTION COL1))
```

The three items, labelled A, B, and C are all declared as **NWAY** items, and are also specified to belong to the same **NWAY** Collection, Col1. This is how a number of **NWAY** items are collected together. The property **NWAYPROPS** (DESELECT T) on the first **NWAY** item specifies that the Col1 Collection is to have the Deselect property enabled. This simply means that the **NWAY** collection can be put in the state where none of the items (A, B, or C) are selected (highlighted). Additionally, each item is declared with a box whose width is one dot (pixel) around it.

The second column in this nested group is specified by:

```
((TYPE STATE LABEL "Choose Me" BOX 1 MENUITEMS (BRAVO DELTA)
     INITSTATE DELTA LINKS (DISPLAY (GROUP ALPHA)))
 (TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35))
```

Column two contains two items, a **STATE** item and a **DISPLAY** item. The **STATE** item is labelled "Choose Me." A Label can be a string or a bitmap, as well as an atom. Selecting the **STATE** item will cause a pop-up menu to appear with two choices for the state of the item, BRAVO and DELTA. The items to go in the pop-up menu are designated by the **MENUITEMS** property.

The pop-up menu would look like:

The initial state of the "Choose Me" item is designated to be DELTA by the **INITSTATE** Item Property. The initial state can be anything; it does not have to be one of the items in the pop-up menu.

Next, the **STATE** item is Linked to a **DISPLAY** item, so that the current state of the item will be displayed in the Free Menu. The link's name is DISPLAY (a special link name for **STATE** items), and the item linked to is described by the Link Description, (GROUP ALPHA). Normally the linked item can just be described by its ID. But in this case, there is more than one item whose ID is ALPHA (for the sake of this example), specifically the first item in the fourth row and the display item in this nested group. The form (GROUP ALPHA) tells Free Menu to search for an item whose ID is ALPHA, limiting the search to the items that are within this lexical group. The lexical group is the smallest group that is declared with the **GROUP** keyword (i.e., not row and column groups) that contains this item declaration. So in this case, Free Menu will link the **STATE** item to the **DISPLAY** item, rather than the first item in the fourth row, since *that* item is outside of the nested group. For further discussion of linking items, see Section 28.7.12, Free Menu Item Links.

Now, establish the **DISPLAY** item:

```
(TYPE DISPLAY ID ALPHA LABEL "" BOX 1 MAXWIDTH 35)
```

We have given it the ID of Alpha that the above**STATE** item uses in finding the proper **DISPLAY** item to link to. This display item is used to display the current state of the item "Choose Me." Every item is required to have a Label property specified, but the label for this **DISPLAY** item will depend on the state of "Choose Me." That is, when the state of the "Choose Me" item is changed from DELTA to BRAVO, the label of the **DISPLAY** item will also change. The null string serves to hold the place for the changeable label.

A box is specified for this item. Since the label is the empty string, Free Menu would draw a very small box. Instead, the **MAXWIDTH** property indicates that the label, whatever it becomes, will be limited to a stringwidth of 35. The width restriction of 35 was chosen because it is big enough for each of the possible labels for this display item. So Free Menu draws the box big enough to enclose any item within this width restriction.

Finally we specify the final item in row four:

```
(LABEL THREE)
```

28.7.4 Free Menu Group Properties

Each group has properties. Most group properties are relevant and should be set in the group's **PROPS** list in the Free Menu Description. User properties can be freely included in the **PROPS** list. A few other properties are set up by the formatter. The macros **FM.GROUPPROP** or **FM.MENUPROP** allow access to group properties after the Free Menu is created.

ID The identifier of this group. Setting the group **ID** is desirable, for example, if the application needs to get handles on items in particular groups, or access group properties.

FORMAT One of **ROW**, **COLUMN**, **TABLE**, or **EXPLICIT**. The default is **ROW**.

FONT A font description of the form **(FAMILY SIZE FACE)**, or a **FONTDESCRIPTOR** data type. This will be the default font for each item in this group. The default font of the top group is the value of the variable **DEFAULTFONT**.

COORDINATES One of **GROUP** or **MENU**. This property applies only to **EXPLICIT** formatting. If **GROUP**, the items in the **EXPLICIT** group are positioned in coordinates relative to the lower left corner of the group, as determined by the mother group. If **MENU**, which is the default, the items are positioned relative to the lower left corner of the menu.

LEFT Specifies a left offset for this group, pushing the group to the right.

BOTTOM Specifies a bottom offset for this group, pushing the group up.

ROWSPACE Specifies the number of dots between rows in this group.

COLUMNSPACE Specifies the number of dots between columns in this group.

BOX Specifies the number of dots in the box around this group of items.

BOXSHADE Specifies the shade of the box.

BOXSPACE Specifies the number of bits between the box and the items.

BACKGROUND The background shade of this group. Nested groups inherit this background shade, but items in this group and nested groups do not. This is because, in general, it is difficult to read text on a background, so items appear on a white background by default. This can be overridden by the **BACKGROUND** Item Property.

28.7.5 Other Group Properties

The following group properties are set up and maintained by Free Menu. The application should probably not change any of these properties.

ITEMS A list of the items in the group.

REGION The region that is the extent of the items in the group.

MOTHER The **ID** of the group that is the mother of this group.

DAUGHTERS A list of **ID** of groups which are daughters to this group.

28.7.6 Free Menu Items

Each Free Menu Item is stored as an instance of the data type **FREEMENUITEM**. Free Menu Items can be thought of as objects, each item having its own particular properties, such as its type, label, and mouse event functions. A number of useful item types, described in Section 28.7.11, Predefined Item Types, are predefined by Free Menu. New types of items can be defined by the application, using Display items as a base. Each Free Menu Item is created from a Free Menu Item Description when the Free Menu is created.

28.7.7 Free Menu Item Descriptions

A Free Menu Item Description is a list in property list format, specifying the properties of the item. For example:

```
(LABEL Refetch SELECTEDFN MY.REFETCHFN)
```

describes a **MOMENTARY** item labelled Refetch, with the function **MY.REFETCHFN** to be called when the item is selected. None of the property values in an item description are evaluated. When constructing Free Menu descriptions that incorporate evaluated expressions (for example labels that are bitmaps) it is helpful to use the backquote facility. For instance, if the value of the variable **MYBITMAP** is a bitmap, then

```
(FREEMENU '(((LABEL A) (LABEL ,MYBITMAP))))
```

would create a Free Menu of one row, with two items in that row, the second of which has the value of **MYBITMAP** as its label.

28.7.8 Free Menu Item Properties

The following Free Menu Item Properties can be set in the Item Description. Any other properties given in an Item Description will be treated as user properties, and will be saved on the **USERDATA** property of the item.

TYPE The type of the item. Choose from one of the Free Menu Item type keywords **MOMENTARY, TOGGLE, 3STATE, STATE, NWAY, EDITSTART, EDIT, NUMBER,** or **DISPLAY**. The default is **MOMENTARY**.

LABEL An atom, string, or bitmap. Bitmaps are always copied, so that the original will not be changed. This property must be specified for every item.

FONT The font in which the item appears. The default is the font specified for the group containing this item. Can be a font description of the form **(FAMILY SIZE FACE)**, or a **FONTDESCRIPTOR** data type.

ID May be used to specify a unique identifier for this item, but is not necessary.

LEFT and **BOTTOM** When **ROW, COLUMN,** or **TABLE** formatting, these specify offsets, pushing the item right and up, respectively, from where the formatter would have put the item. In **EXPLICIT** formatting, these are the actual coordinates of the item, in the coordinate system given by the group's **COORDINATES** property.

HJUSTIFY Indicates horizontal justification type: **LEFT, CENTER,** or **RIGHT**. Specifies that this item is to be horizontally justified within the extent of its group. Note that the main group, as opposed to the smaller row or column group, is used.

VJUSTIFY Specifies that this item is to be vertically justified. Values are **TOP, MIDDLE,** or **BOTTOM**.

HIGHLIGHT Specifies the highlighted looks of the item, that is, how the item changes when a mouse event occurs on it. See Section 28.7.12, Free Menu Item Highlighting, for more details on highlighting.

MESSAGE Specifies a string that will be printed in the prompt window after a mouse cursor selects this item for **MENUHELDWAIT** milliseconds. Or, if an atom, treated as a function to get the message. The function is passed three arguments, **ITEM, WINDOW,** and **BUTTONS**, and should return a string. The default is a message appropriate to the type of the item.

INITSTATE Specifies the initial state of the item. This is only appropriate to **TOGGLE, 3STATE,** and **STATE** items.

MAXWIDTH Specifies the width allowed for this item. The formatter will leave enough space after the item for the item to grow to this width without collisions.

MAXHEIGHT Similar to **MAXWIDTH**, but in the vertical dimension.

BOX Specifies the number of bits in the box around this item. Boxes are made around **MAXWIDTH** and **MAXHEIGHT** dimensions. If unspecified, no box is drawn.

BOXSHADE Specifies the shade that the box is drawn in. The default is BLACKSHADE.

BOXSPACE Specifies the number of bits between the box and the label. The default is one bit.

BACKGROUND Specifies the background shade on which the item appears. The default is WHITESHADE, regardless of the group's background.

LINKS Can be used to link this item to other items in the Free Menu. See Section 28.7.13, Free Menu Item Links, for more information.

28.7.9 Mouse Properties

The following properties provide a way for application functions to be called under certain mouse events. These functions are called with the **ITEM**, the **WINDOW**, and the **BUTTONS** passed as arguments. These application functions do not interfere with any Free Menu system functions that take care of handling the different item types. In each case, though, the application function is called after the system function. The default for all of these functions is **NILL**. The value of each of the following properties can be the name of a function, or a lambda expression.

SELECTEDFN Specifies the function to be called when this item is selected. The **Edit** and **EditStart** items cannot have a **SELECTEDFN**. See the Edit Free Menu item description in Section 28.7.11, Predefined Item Types, for more information.

DOWNFN Specifies the function to be called when the item is selected with the mouse cursor.

HELDFN Specifies the function to be called repeatedly when the item is selected with the mouse cursor.

MOVEDFN Specifies the function to be called when the mouse cursor moves off this item (mouse buttons are still depressed).

28.7.10 System Properties

The following Free Menu Item properties are set and maintained by Free Menu. The application should probably not change these properties directly.

GROUPID Specifies the **ID** of the smallest group that the item is in. For example, in a row formatted group, the item's **GROUPID** will be set to the **ID** of the row that the item is in, not the **ID** of the whole group.

STATE Specifies the current state of **TOGGLE**, **3STATE**, or **STATE** items. The state of an **NWAY** item behaves like that of a toggle item.

BITMAP Specifies the bitmap from which the item is displayed.

REGION Specifies the region of the item, in window coordinates. This is used for locating the display position, as well as determining the mouse sensitive region of the item.

MAXREGION Specifies the maximum region the item may occupy, determined by the **MAXWIDTH** and **MAXHEIGHT** properties (see Section

28.7.8, Free Menu item Properties). This is used by the formatter and the display routines.

SYSDOWNFN
SYSMOVEDFN
SYSSELECTEDFN

These are the system mouse event functions, set up by Free Menu according to the item type. These functions are called before the mouse event functions, and are used to implement highlighting, state changes, editing, etc.

USERDATA Specifies how any other properties are stored on this list in property list format. This list should probably not need to be manipulated directly.

28.7.11 Predefined Item Types

MOMENTARY [Free Menu Item]

MOMENTARY items are like command buttons. When the button is selected, its associated function is called.

TOGGLE [Free Menu Item]

Toggle items are simple two-state buttons. When pressed, the button is highlighted; it stays that way until pressed again. The states of a toggle button are **T** and **NIL**; the initial state is **NIL**.

3STATE [Free Menu Item]

3STATE items rotate through **NIL**, **T**, and **OFF**, states each time they are pressed. The default looks of the **OFF** state are with a diagonal line through the button, while **T** is highlighted, and **NIL** is normal. The default initial state is **NIL**.

The following Item Property applies to **3STATE** items:

OFF Specifies the looks of a **3STATE** item in its **OFF** state. Similar to **HIGHLIGHT**. The default is that the label gets a diagonal slash through it.

STATE [Free Menu Item]

STATE items are general multiple state items. The following Item Property determines how the item changes state:

CHANGESTATE This Item Property can be changed at any time to change the effect of the item. If a **MENU** data type, this menu pops up when the item is selected, and the user can select the new state. Otherwise, if this property is given, it is treated as a function name, which is passed three arguments, **ITEM**, **WINDOW**, and **BUTTONS**. This function can do whatever it wants, and is expected to return the new state (an atom, string, or bitmap), or **NIL**, indicating the state should not change. The state of the item can automatically be indicated in the Free Menu, by setting up a **DISPLAY** link to a **DISPLAY** item in the menu (see Section 28.7.13, Free Menu Item Links). If such a link exists, the label of the **DISPLAY** item will be changed to the new state. The possible states are not restricted at all, with the exception of selections from a pop-up menu. The state can be changed to any atom, string, or bitmap, manually via **FM.CHANGESTATE**.

The following Item Properties are relevant to **STATE** items when building a Free Menu:

MENUITEMS If specified, should be a list of items to go in a pop-up menu for this item. Free Menu will build the menu and save it as the **CHANGESTATE** property of the item.

MENUFONT The font of the items in the pop-up menu.

MENUTITLE The title of the pop-up menu. The default title is the label of the **STATE** item.

NWAY [Free Menu Item]

NWAY items provide a way to collect any number of items together, in any format within the Free Menu. Only one item from each Collection can be selected at a time, and that item is highlighted to indicate this. The following Item Properties are particular to **NWAY** items:

COLLECTION An identifier that specifies which **NWAY** Collection this item belongs to.

NWAYPROPS A property list of information to be associated with this collection. This property is only noticed in the Free Menu Description on the first item in a **COLLECTION**. **NWAY** Collections are formed by creating a number of **NWAY** items with the same **COLLECTION** property. Each **NWAY** item acts individually as a Toggle item, and can have its own mouse event functions. Each **NWAY** Collection itself has properties, its state for instance. After the Free Menu is created, these Collection properties can be accessed by the macro **FM.NWAYPROPS**. Note that **NWAY** Collections are different from Free Menu Groups. There are three **NWAY** Collection properties that Free Menu looks at:

DESELECT If given, specifies that the Collection can be deselected, yielding a state in which no item in the Collection is selected. When this property is set, the Collection can be deselected by selecting any item in the Collection and pressing the right mouse button .

STATE The current state of the Collection, which is the actual item selected.

INITSTATE Specifies the initial state of the Collection. The value of this property is an Item Link Description (see Section 28.7.13, Free Menu Item Links.)

EDIT [Free Menu Item]

EDIT items are textual items that can be edited. The label for an **EDIT** item cannot be a bitmap. When the item is selected an edit caret appears at that cursor position within the item, allowing insertion and deletion of characters at that point. If selected with the right mouse button, the item is cleared before editing starts. While editing, the left mouse button moves the caret to a new position within the item. The right mouse button deletes from the caret to the cursor. CONTROL-W deletes the previous word. Editing is stopped when another item is selected, when the user moves the cursor into another TTY window and clicks the cursor, or when the Free Menu function **FM.ENDEDIT** is called (called when the Free Menu is reset, or the window is closed). The Free Menu editor will time out after about a minute, returning automatically. Because of the many ways in which editing can terminate, **EDIT** items are not allowed to have a **SELECTEDFN**, as it is not clear when this function should be called. Each **EDIT** item should have an ID specified, which is used when getting the state of the Free Menu, since the string being edited is defined as the state of the item, and thus cannot distinguish edit items. The following Item Properties are specific to **EDIT** items.

MAXWIDTH Specifies the maximum string width of the item, in bits, after which input will be ignored. If **MAXWIDTH** is not specified, the items becomes infinitely wide and input is never restricted.

INFINITEWIDTH This property is set automatically when **MAXWIDTH** is not specified. This tells Free Menu that the item has no right end, so that the item becomes mouse sensitive from its left edge to the right edge of the window, within the vertical space of the item.

LIMITCHARS The input characters allowed can be restricted in two ways: If this item property is a list, it is treated as a list of legal characters; any character not in the list will be ignored. If it is an atom, it is treated as the name of a test predicate, which is passed three arguments, **ITEM**, **WINDOW**, and **CHARACTER**, when each character is typed. This predicate should return **T** if the character is legal, **NIL** otherwise. The **LIMITCHARS** function can also call **FM.ENDEDIT** to force the editor to terminate, or **FM.SKIPNEXT**, to cause the editor to jump to the next edit item in the menu.

ECHOCHAR This item property can be set to any character. This character will be echoed in the window, regardless of what character is typed. However the item's label contains the actual string typed. This is useful for operations like password prompting. If **ECHOCHAR** is used, the font of the item must be fixed pitch. Unrestricted **EDIT** items should not have other items to their right in the menu, as they will be replaced. If the item is boxed, input is restricted to what will fit in the box. Typing off the edge of the window will cause the window to scroll appropriately. Control characters can be edited, including the carriage return and line feed, and they are echoed as a black box. While editing, the Skip/Next key ends editing the current item, and starts editing the next **EDIT** item in the Free Menu.

NUMBER [Free Menu Item]

NUMBER items are **EDIT** items that are restricted to numerals. The state of the item is coerced to the the number itself, not a string of numerals. There is one **NUMBER**- specific Item Property:

NUMBERTYPE If **FLOATP** (or **FLOAT**), then decimals are accepted. Otherwise only whole numbers can be edited.

EDITSTART [Free Menu Item]

EDITSTART items serve the purpose of starting editing on another item when they are selected. The associated Edit item is linked to the EditStart item by an **EDIT** link (see Free Menu Item Links below). If the **EDITSTART** item is selected with the right mouse button, the Edit item is cleared before editing is started. Similar to **EDIT** items, **EDITSTART** items cannot have a **SELECTEDFN**, as it is not clear when the associated editing will terminate.

DISPLAY [Free Menu Item]

DISPLAY items serve two purposes. First, they simply provide a way of putting dummy text in a Free Menu, which does nothing when selected. The item's label can be changed, though. Secondly, **DISPLAY** items can be used as the base for new item types. The application can create new item types by specifying **DOWNFN**, **HELDFN**, **MOVEDFN**, and **SELECTEDFN** for a **DISPLAY** item, making it behave as desired.

28.7.12 Free Menu Item Highlighting

Each Free Menu Item can specify how it wants to be highlighted. First of all, if the item does not specify a **HIGHLIGHT** property, there are two default highlights. If the item is not boxed, the label is simply inverted, as in normal menus. If the item is boxed, it is highlighted in the shade of the box. Alternatively, the value of the **HIGHLIGHT** property can be a SHADE, which will be painted on top of the item when a mouse event occurs on it. Or the **HIGHLIGHT** property can be an alternate label, which can be an atom, string or bitmap. If the highlight label is a different size than the item label, the formatter will leave enough space for the larger of the two. In all of these cases, the looks of the highlighted item are determined when the Free Menu is built, and a bitmap of the item with these looks is created. This bitmap is stored on the item's **HIGHLIGHT** property, and simply displayed when a mouse event occurs. The value of the highlight property in the Item Description is copied to the **USERDATA** list, in case it is needed later for a label change.

28.7.13 Free Menu Item Links

Links between items are useful for grouping items in abstract ways. In particular, links are used for associating **EDITSTART** items with their item to edit, and **STATE** items with their state display. The Free Menu Item property **LINKS** is a property list, where the value of each Link Name property is a pointer to another item. In the Item Description, the value of the **LINK** property should be a property list as above. The value of each

Link Name property is a Link Description. A Link Description can be one of the following forms:

<ID> An ID of an item in the Free Menu. This is acceptable if items can be distinguished by ID alone.

(<GROUPID> <ID>) A list whose first element is a GROUPID, and whose second element is the ID of an item in that group. This way items with similar purposes, and thus similar ID's, can be distinguished across groups.

(GROUP <ID>) A list whose first element is the keyword GROUP, and whose second element is an item ID. This form describes an item with ID, in the same group that this item is in. This way you do not need to know the GROUPID, just which group it is in.

Then after the entire menu is built, the links are set up, turning the Link Descriptions into actual pointers to Free Menu Items. There is no reason why circular Item Links cannot be created, although such a link would probably not be very useful. If circular links are created, the Free Menu will not be garbage collected after it is not longer being used. The application is responsible for breaking any such links that it creates.

28.7.14 Free Menu Window Properties

FM.PROMPTWINDOW Specifies the window that Free Menu should use for displaying the item's messages. If not specified, **PROMPTWINDOW** is used.

FM.BACKGROUND The background shade of the entire Free Menu. This property can be set automatically by specifying a **BACKGROUND** argument to the function **FREEMENU**. The window border must be 4 or greater when a Free Menu background is used, due to the way the Window System handles window borders.

FM.DONTRESHAPE Normally, Free Menu will attempt to use empty space in a window by pushing items around to fill the space. When a Free Menu window is reshaped, the items are repositioned in the new shape. This can be disabled by setting the **FM.DONTRESHAPE** window property.

28.7.15 Free Menu Interface Functions

(FREEMENU *DESCRIPTION TITLE BACKGROUND BORDER***)** [Function]

Creates a Free Menu from a Free Menu Description, returning the window. This function will return quickly unless new display fonts have to be created.

28.7.16 Accessing Functions

(FM.GETITEM *ID GROUP WINDOW***)** [Function]

Gets item *ID* in *GROUP* of the Free Menu in *WINDOW*. This function will search the Free Menu for an item whose *ID* property matches, or secondly whose **LABEL** property matches *ID*. If *GROUP* is **NIL**, then the entire Free Menu is searched. If no matching item is found, **NIL** is returned.

(FM.GETSTATE *WINDOW*) [Function]

Returns in property list format the ID and current **STATE** of every **NWAY** Collection and item in the Free Menu. If an item's or Collection's state is **NIL**, then it is not included in the list. This provides an easy way of getting the state of the menu all at once. If the state of only one item or Collection is needed, the application can directly access the **STATE** property of that object using the Accessing Macros described in Section 28.7.20, Free Menu Macros. This function can be called when editing is in progress, in which case it will provide the label of the item being edited at that point.

28.7.17 Changing Free Menus

Many of the following functions operate on Free Menu Items, and thus take the item as an argument. The *ITEM* argument to these functions can be the Free Menu Item itself, or just a reference to the item. In the second case, **FM.GETITEM** (see Section 28.7.16, Accessing Functions) will be used to find the item in the Free Menu. The reference can be in one of the following forms:

<ID> Specifies the first item in the Free Menu whose ID or LABEL property matches <ID>.

(<GROUPID> <ID>) Specifies the item whose ID or LABEL property matches <ID> within the group specified by <GROUPID>.

(FM.CHANGELABEL *ITEM NEWLABEL WINDOW UPDATEFLG*) [Function]

Changes an *ITEM*'s label after the Free Menu has been created. It works for any type of item, and **STATE** items will remain in their current state. If the window is open, the item will be redisplayed with its new appearance. *NEWLABEL* can be an atom, a string, or a bitmap (except for **EDIT** items), and will be restricted in size by the **MAXWIDTH** and **MAXHEIGHT** Item Properties. If these properties are unspecified, the *ITEM* will be able to grow to any size. *UPDATEFLG* specifies whether or not the regions of the groups in the menu are recalculated to take into account the change of size of this item. The application should not change the label of an **EDIT** item while it is being edited. The following Item Property is relevant to changing labels:

CHANGELABELUPDATE Exactly like **UPDATEFLG** except specified on the item, rather than as a function paramater.

(FM.CHANGESTATE *X NEWSTATE WINDOW*) [Function]

Programmatically changes the state of items and **NWAY** Collections. *X* is either an item or a Collection name. For items *NEWSTATE* is a state appropriate to the type of the item. For **NWAY** Collections, *NEWSTATE* should be the desired item in the Collection, or **NIL** to deselect. For **EDIT** and **NUMBER** items, this function just does a label change. If the window is open, the item will be redisplayed.

(FM.RESETSTATE *ITEM WINDOW*) [Function]

Sets an *ITEM* back to its initial state.

(FM.RESETMENU *WINDOW***)** [Function]

Resets every item in the menu back to its initial state.

(FM.RESETSHAPE *WINDOW ALWAYSFLG***)** [Function]

Reshapes the *WINDOW* to its full extent, leaving the lower-left corner unmoved. Unless *ALWAYSFLG* is **T**, the window will only be increased in size as a result of resetting the shape.

(FM.RESETGROUPS *WINDOW***)** [Function]

Recalculates the extent of each group in the menu, updating group boxes and backgrounds appropriately.

(FM.HIGHLIGHTITEM *ITEM WINDOW***)** [Function]

Programmatically forces an *ITEM* to be highlighted. This might be useful for *ITEM*s which have a direct effect on other *ITEM*s in the menu. The *ITEM* will be highlighted according to its **HIGHLIGHT** property, as described in Section 28.7.12, Free Menu Item Highlighting. This highlight is temporary, and will be lost if the *ITEM* is redisplayed, by scrolling for example.

28.7.18 Editor Functions

(FM.EDITITEM *ITEM WINDOW CLEARFLG***)** [Function]

Starts editing an **EDIT** or **NUMBER** *ITEM* at the beginning of the *ITEM*, as long as the *WINDOW* is open. This function will most likely be useful for starting editing of an *ITEM* that is currently the null string. If *CLEARFLG* is set, the *ITEM* is cleared first.

(FM.SKIPNEXT *WINDOW CLEARFLG***)** [Function]

Causes the editor to jump to the beginning of the next **EDIT** item in the Free Menu. If *CLEARFLG* is set, then the next item will be cleared first. If there is not another **EDIT** item in the menu, this function will simply cause editing to stop. If this function is called when editing is not in progress, editing will begin on the first **EDIT** item in the menu. This function can be called from any process, and can also be called from inside the editor, in a **LIMITCHARS** function.

(FM.ENDEDIT *WINDOW WAITFLG***)** [Function]

Stop any editing going on in *WINDOW*. If *WAITFLG* is **T**, then block until the editor has completely finished. This function can be called from another process, or from a **LIMITCHARS** function.

(FM.EDITP *WINDOW***)** [Function]

If an item is in the process of being edited in the Free Menu *WINDOW*, that item is returned. Otherwise, **NIL** is returned.

28.7.19 Miscellaneous Functions

(FM.REDISPLAYMENU *WINDOW***)** [Function]

Redisplays the entire Free Menu in its *WINDOW*, if the *WINDOW* is open.

(FM.REDISPLAYITEM *ITEM WINDOW***)** [Function]

Redisplays a particular Free Menu *ITEM* in its *WINDOW*, if the *WINDOW* is open.

(FM.SHADE *X SHADE WINDOW***)** [Function]

X can be an item, or a group ID. *SHADE* is painted on top of the item or group. Note that this is a temporary operation, and will be undone by redisplaying. For more permanent shading, the application may be able to add a **REDEDISPLAYFN** and **SCROLLFN** for the window as necessary to update the shading.

(FM.WHICHITEM *WINDOW POSorX Y***)** [Function]

Locates and identifies an item from its known location within the *WINDOW*. If *WINDOW* is **NIL**, **(WHICHW)** is used, and if *POSorX* is **NIL**, the current cursor location is used.

(FM.TOPGROUPID *WINDOW***)** [Function]

Return the ID of the top group of this Free Menu.

28.7.20 Free Menu Macros

These Accessing Macros are provided to allow the application to get and set information in the Free Menu data structures. They are implemented as macros so that the operation will compile into the actual access form, rather than figuring that out at run time.

(FM.ITEMPROP *ITEM PROP {VALUE}***)** [Macro]

Similar to **WINDOWPROP**, this macro provides an easy access to the fields of a Free Menu Item. The function **FM.GETITEM** gets the *ITEM*, described in Section 28.7.16, Accessing Function. *VALUE* is optional, and if not given, the current value of the *PROP* property will be returned. If *VALUE* is given, it will be used as the new value for that *PROP*, and the old value will be returned. When a call to **FM.ITEMPROP** is compiled, if the *PROP* is known (quoted in the calling form), the macro figures out what field to access, and the appropriate Data Type access form is compiled. However, if the *PROP* is not known at compile time, the function **FM.ITEMPROP**, which goes through the necessary property selection at run time, is compiled. The **TYPE** and **USERDATA** properties of a Free Menu Item are Read Only, and an error will result from trying to change the value of one of these properties.

(FM.GROUPPROP *WINDOW GROUP PROP {VALUE}***)** [Macro]

Provides access to the Group Properties set up in the *PROPS* list for each group in the Free Menu Description. *GROUP* specifies the ID of the desired group, and *PROP* the name of the desired property. If *VALUE* is specified, it will become the new value of the property, and the old value will be returned. Otherwise, the current value is returned.

(FM.MENUPROP *WINDOW PROP {VALUE}*) [Macro]

Provides access to the group properties of the top-most group in the Free Menu, that is to say, the entire menu. This provides an easy way for the application to attach properties to the menu as a whole, as well as access the Group Properties for the entire menu.

(FM.NWAYPROP *WINDOW COLLECTION PROP {VALUE}*) [Macro]

This macro works just like **FM.GROUPPROP**, except it provides access to the **NWay** Collections.

[This page intentionally left blank]